Designing and synthesizing the synchronization of concurrent object-oriented systems

Ph.D. Thesis

Szabolcs Hajdara

Supervisor:
László Kozma, C.Sc.

Ph.D. Program in Informatics
Ph.D. School of Informatics
Prof. János Demetrovics

Eötvös Loránd University
Faculty of Informatics
Department of Software Technology and Methodology

Budapest, 2007

Supported by GVOP-3.2.2-2004-07-005/3.0.

Contents

Chapter 1

Introduction

Nowadays object-oriented programming is a very important research area in informatics. One of the most important group of object-oriented languages is the group of class based imperative languages. There are very useful program designing methodologies (e.g. Rational Unified Process – RUP [74]) based on object-oriented techniques that are supported by CASE tools (e.g. Rational Rose [108], Software Through Pictures [61], [104], etc.) and other programs that help in designing systems, and so the design and development of programs become easy and safe. Object-oriented software development methodologies make the design and programming easier and faster by bringing them closer to the real problem. Furthermore, customers can better understand the steps and the partial results of software developing so the product becomes more useful. Systems that are built on object-oriented technologies have several advantages, therefore it is worth trying to make this methodology more efficient.

The great improvement of the hardware, which is experienced in the last years, resulted significant changes in the environment in which we run our software. Software run more and more in a hardware environment with more processors, or in many cases connected computers are used. So designing the synchronization of entities gets larger and larger role in the object-oriented software development. Nowadays it is very popular to use distributed systems in informatics. The improvement and spread of the Java programming language (and so the JRMI – Java Remote Method Invocation – technique)

[44], [45], [46], [64], [102], the J2EE technology [103], the CORBA standard [60], [62], [107], DCOM [48], [116], .NET [1], [20] and other software tools caused that developing distributed object-oriented software has become commonplace. That is why the improvement of object-oriented parallel designing techniques is so important.

The correctness of object-oriented programs can usually be proven with hard work, so the dissection of the proper functioning of the synchronization code of an object is also very difficult. Moreover, writing correct parallel programs requires much more attention, even small imprecisions may cause serious failures or the blocking of the system. Our observation is that methods that prove correctness of programs with behavioural analysis [117] are much more sensitive in the case of parallel programs. This fact shows that parallel programs are less clear for programmers. Therefore, developing tools that help writing of correct programs is a natural demand.

One of the greatest problem of writing parallel programs is to make a correct and easy modifiable synchronization code. Synchronization code and computation code of programs can usually be separated (see [7]). We consider only the synchronization code of objects and examine how to connect the synchronization code to the computation code, but we do not discuss the computation code itself. We aim to produce (surely) correct synchronization code for objects.

Correctnesses of programs depend on the requirements of the customers. So a usable specification is needed so that the correctness can be proved. There are four main methods of creating proven correct programs: synthesizing a correct program form the specification (in this case, the correctness of the synthesis method is proven) [4], [7], [38], [81], [85], [87], [88], [89], giving a solution followed by a proof of the correctness of the solution based on the specification [65], [72], [105], using a model checker [12], [21], [27], [93], [125] for an automatic proof of the correctness, or testing the program [14], [41], [49], [113]. These approaches have advantages and disadvantages. For example, if a synthesis algorithm is used, it is not needed to code some parts of the program by hand, but no additional knowledge can be applied. In the other cases, program code has to be written by programmers, but humans

may exploit the specialities of the system.

As we mentioned, we concentrate only for the synchronization part of programs and we have chosen to use synthesis algorithms to produce the synchronization code. We will examine how to design and synthesize the synchronization of an object-oriented system. Synchronization and computation parts of objects cannot always be entirely separated, because synchronization depends on the computation. Synchronization is needed, because the computation is in a state where it needs to be synchronized with the others (e.g. a resource has to be used and the resource can be used by only one object at a time, etc.). If the computation part of an object is in a state where it needs to change the synchronization state, it calls the synchronization code. The connection of these two parts of the code has to be specified, for which we have given a method, furthermore, the method generates the synchronization calls automatically.

We consider only imperative class based object-oriented programming languages. In such cases, the synchronization of objects is implemented in classes. There are methods that are developed for specifying and synthesizing the synchronization code of processes (e.g. in [7], [38]). We chose a method and extended it so that it could be used for object-oriented systems. We had to extend the specification method so that we could design an object-oriented synchronization and we gave an algorithm that produces the concrete object-oriented synchronization code. Furthermore, our algorithm creates a skeleton of the computation code where synchronization calls are invoked.

A synthesized code is usually not optimal. Synthesis algorithms are working with general rules so that they can work in most cases, but in many cases if some special knowledge were used, the generated code would be more efficient. There are cases when it is possible to optimize the synthesized synchronization code. We shortly examine the possibilities of optimization in certain special cases.

In the case of object-oriented systems, there are problems that have to be handled. For example, reusing the program code is a very useful tool that can be used in an object-oriented environment, but as it is known, reusing the synchronization code can cause problems [16], [26], [92], [127].

Unfortunately, finding a general solution for these problems is very difficult, however we examine the problem and provide a solution for some special cases.

We will consider special examples, where our original method cannot be applied and we show a modification of the original method that extends the usability of our algorithm.

If an object-oriented system is to be constructed, we can use various tools to design the program. It is a natural requirement to plan the specification language of the synchronization so that it can be used together with a widely used tool. These tools usually support using UML (see [90], [135]) to design object-oriented systems. We give a method with which the synchronization of a system can be designed similarly to the sequential parts, so it is easier to learn.

This thesis is organized as follows. In Chapter 2, we begin with a short overview of the object-oriented programming and the problems of parallel programs. Furthermore, a summary is shown of techniques we have built our algorithms upon. In Chapter 3, we give a method, with which synchronization code of parallel object-oriented systems can be automatically generated. First we describe simpler problems, then extend the method for more complex cases, and give an algorithm to handle the connections of parallel running objects. We deal with designing shared classes and the problems of the inheritance. In Chapter 4, some extensions of our synthesis algorithm are shown, and a graphical representation of the specification is also given to be able to embed it in a UML CASE tool. In Chapter 5, we refer to the related works and show the differences. In Chapter 6, we summarize our results and in Chapter 7, we show the main problems we aim to solve in the future.

I have done my researches together with Balázs Ugron, we have published our results together. After labouring the bases, Balázs Ugron continued his work with pipeline systems [129], while I continued my work in the area of the object-oriented parallel systems.

Chapter 2

Background

We will consider a technique, how to design parallelism of object-oriented (OO) systems. Of course, we want to use every beneficial properties of object-oriented systems and build our method upon the strong building blocks that was developed for parallel technologies. Furthermore, we have to take care of the main difficulties, which can be arisen by working with objects and we want to avoid the problems of concurrent processes, if it is possible. So we give a short overview of object-oriented systems and specifying and synthesizing the synchronization code of processes, and we examine on the well known problems of object-oriented parallel systems.

2.1 Object-oriented systems

Object-oriented systems are designed to be composed of objects and object-oriented languages are used for implementing them. If a class based language is used for developing an object-oriented system, the objects are instances of classes. There are actor based languages (e.g. POOL-T [3]) where there are not classes, only actors. We consider only class based languages in the following. So if we build an object-oriented system, we design the system in a class based way and use a language for coding where classes, and instances of classes (objects) can be used as building blocks. Object-oriented programming has several advantages, because developers can use techniques

that can help very much in developing programs, so reliable software can be made by using object-oriented tool. For example, inheritance makes software developing faster and gives the possibility of building our applications on safe components. Furthermore, object-oriented programs are more understandable, so programmers can maintain them better.

Developing object-oriented software has a further advantage, namely developers can use well known graphical CASE tools for designing the software, moreover, the used graphical diagrams are understandable even for customers, viz. it models the system in the problem space.

2.1.1 Object-oriented software technology

As we mentioned before, we will consider building object-oriented systems. Developing a software system is a special procedure, because a software is used for developing an other software. Software developing needs usable techniques, therefore, every improvement of the existing techniques or intruducing new ones can help to make softvare development safer.

Software developers want to make quality products. Quality means that the program provides all the functonalities that are required by the customer without errors. Because the developer and the user of a program are commonly different persons, there is an elementary error source during the development, because designers often do not understand customers. Therefore the product contain errors. These errors must be corrected, meanwhile the software must be improved. This – usually livelong and expensive – phase is the maintenance of software and it requires automatisms to keep the expenses down.

It is expected from the programming method which is applied during the life cycle of a program to be usable for solving the problems figured by the users and providing the possibilities of improvement and extendibility of the provided services. According to these, the methodology of the software technology are built upon the abstraction and modularization, which is written in [51]. By using object-oriented software-development techniques, modularization results classes of objects.

Many engineers and developers use already existing schemas of methods to make their work more efficient, therefore, it is important to improve the useful methods and introduce new ones.

An advanced approach of software development is development with creating models. By using this approach, a prototype of the solution is created in the early phase of the development. There are several modelling languages (e.g. [136]), however, maybe the most widespread is UML (see [15], [90], [135]). During modelling, the object-oriented disciplines are followed. Software engineers analyse the model, whether it corresponds to the requirement specification. So, if a model is created and analysed, then after some modification steps it can be assumed that the model is correct. If we have a correct model, the more parts of the program code can be generated the more improvement of the develepment can be achieved. That is why we concentrate on specifying some parts of the program, which can be used to generate the corresponding parts of the source code later.

We decided to follow model creating approach, and we will start with a UML model at the most abstract level and we concentrate only on the synchronization part of programs.

You can read more about software technologies in [8], [47], [67], [73], [118] and [119].

2.1.2 Object-oriented programming

The reason, why we can design object-oriented programs, is that programs can be written in object-oriented way. So first we have to define, what object-oriented programming means.

Peter Wegner defined object-oriented programming (see [140]) in the following way:
Object-oriented = data abstraction + abstract data type + type inheritance. Data abstraction means a presence of the data that is independent from the concrete syntax and semantic. Abstract data types are presences of the data types that are independent from the implementation. Type inheritance means, that the formal descriptions are reused by mechanically transforma-

tions in order that errors can be avoided.

In a practical approach, object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

2.1.3 Object-oriented design

Object-oriented design is part of OO methodology and it forces programmers to think in terms of objects, rather than procedures, when they design their code. An object contains encapsulated data and procedures that grouped together to represent an entity. The 'object interface', how the object can be interacted, is also defined. An object-oriented program is described by the interaction of the objects. Object-oriented design is the discipline of defining the objects and their interactions to solve a business problem that was identified and documented during the object-oriented analysis. You can read about object-oriented design in [22], [24], [78], [94], [112], [115], [136] and [138].

Object-oriented design includes an analysis step. In fact, analysis is the first step of designing systems. During object-oriented analysis, there is an emphasis on finding and describing the objects – or concepts – in the problem domain, and defining how they collaborate to fulfil the requirements. For this purpose, most object-oriented design tools work with visual diagrams. For example, a Plane software object may have a "tailNumber" attribute and a "getFlightHistory" method. So, we have to visualize that there is a Plane object and it has the corresponding attributes and methods. Furthermore, we know that there are many planes, so we have to use a Plane type and several planes can be instantiated from this. So, by designing an object-oriented system, we have to identify the objects and rate them to object classes.

Designing a system by using object-oriented design, a complex analysis has to be fulfilled, which is followed by a designing part that is based on the analysis. Analysing and designing are usually well visualized, so customers

can be active participants of the process and the result is more understandable to the designers as well, so it is easier to modify them. The common notations to illustrate the features of the system are the diagrams of UML. During the analysis, several diagrams can be made (e.g. use case diagrams in the requirements analysis, interaction diagrams – sequence diagrams and collaboration diagrams –, etc.).

In addition to a dynamic view of collaborating objects that are shown in interaction diagrams, a static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

Programmers have to describe the dynamic behaviour of an object, which shows state changes of the object. This can be done by using state diagrams. Each class can have a state diagram, which defines the life cycle of its objects.

By using object-oriented design, a further advantage can be used. Designers can exploit the power of inheritance, which makes designing and developing software easier. It can be assumed that designers and programmers can use a large object library. Fortunately, there are various kinds of object library collections. For example, most program languages have utility libraries, but if a software developer group makes a kind of programs, e.g. telecommunication programs, then it can develop its own object library that can be used in every development. By using inheritance, these components can be reused by adding new services to them or by modifying some of their methods. This makes software development faster and safer (if the used components are correct). Furthermore, if class B inherits from class A, then B provides all the services of A (perhaps in a modified form). So instances of B can be used if instances of A are required, which makes it easier to define the requirements of the parameters of functions and so the fast replacement of services.

The static skeleton of the program can be built from most of the object-oriented models. It can be used when designing a method to improve the object-oriented development.

You can gain further information about object-oriented software development in [23], [73], [86], [91] and [136].

2.2 Parallel systems

Parallel programs are special programs that run in a concurrent environment [83]. There can be many processors and each processor may execute a program, but programs can run in parallel also in only one processor by using a scheduler (see [35], [120], [144]). Concurrently running programs (processes) can use common resources. Reaching the common resources (shared memory blocks, printers, etc.) has to be synchronized so that processes are not allowed to use a resource if it cannot serve more requests and an other process uses it, furthermore, processes have to wait for each other if they want to use a particular result of an other one.

Parallel programs have to be written so that they work correctly, so tools have to be provided that are able to specify the requirements of running processes in a concurrent environment. Furthermore, processes may use components that have to be designed to be easily usable for the parallel programs. For example, a stack type can be designed in a sequential environment so that it provides the following two operations: top, pop. Operation top returns the element on the top of the stack and pop removes the top element of the stack. This type can be used well in a sequential environment. But, if parallel programs are using this type, it could cause problems: it can happen that a process reads the top element and then removes it while an other process does it too, so it could happen that the top element is read twice but two elements are removed from the stack and the second one was not read. Therefore, in parallel environments the following solution might be better: the pop operation returns the top element and removes it in one step (if there are no elements in the stack, it should denote the fact with a special return value).

It can be seen that the structure of the components and processes is a dominant point of designing parallel systems. So, if the functions are designed so that their concept suits well into the concept of the environment, then synchronizing their running is easier. Considering the example above, if reaching and removing the top element of the stack are separated, then synchronization has to be written in the program that uses the stack instance.

In the other case, synchronization can be written on type level, inside the implementation of the stack. Of course, implementation can be written in the stack type by choosing the first structure too, but in that case we have to introduce other functions, e.g. lock and unlock. In this case, reading and removing the top element of the stack can be achieved by the following sequence of functions: lock; top; pop; unlock. It can be seen that this structure is more flexible (element can be read without removing) but not so safe to use: if the programmer forgets to lock the object, executing of functions may be unsafe. That is why in the case of parallel programs, functions may check – in exclusive mode – whether their preconditions are fulfilled and if it is not, they can throw e.g., an exception.

If n processes are running and there are at least n processors in the computer, then all the processes are running concurrently (real parallelism). If there are less processors, then not all the processes can run in a time, a scheduling algorithm is needed. There are non-pre-emptive schedulers that only schedules the next process if the actual running process yields its executing, is blocked (e.g. because of an I/O operation), or is terminated. By using non-pre-emptive schedulers it is easier to design the system, because designers surely know that executing of a block cannot be interrupted if there are no statements in the program block that can be blocked and they does not write directly yield statements in the code. In the other hand, if a process is waiting for a result of an other process, the first one has to yield its time to give itself the chance to continue its running. If a process is scheduled by a pre-emptive scheduler, its running can be interrupted any time, so designers have to handle this fact. In this case, if a process waits for a result of an other process, it is ensured, that the other process can produce its result without the first one yields its running time. Although, by using non-pre-emptive schedulers may bring on problems of yielding running time, pre-emptive schedulers require more accurate design of programs, and it is more difficult, so we will assume that schedulers are pre-emptive. Besides, we do not consider whether processes have to yield their time because we use techniques that block processes when they wait for data, and if a process is blocked, even non-pre-emptive schedulers run an other process. There

are several subtypes of schedulers according to the algorithm that is used for choosing the next process to run and giving time-slices to processes (e.g. Round-robin, priority scheduling, real-time scheduling, etc. – see [35], [120], [144]).

When running processes concurrently so that processes can use common resources, it can happen that process p reserves resource A, and process q reserves resource B. After that process p requires resource B but it keeps resource A reserved and process q requires resource A while keeping resource B reserved. This is the case of deadlock. Deadlock can happen in a concurrent environment if processes can require resources while keeping reserved others and a circle can evolve in the waiting queue of the processes. Possibility of deadlock is depend on the design of the system. If the system is designed so that every process releases its resources if it has to wait for an other one, then deadlock cannot evolve in the system. Unfortunately, in such cases starvation can appear. Starvation can be arisen if a group of processes can hold resources so that an other group of processes can never catch them. In these cases, a group of processes starves. An other interesting problem of parallel systems is the case of livelock. Livelock is a special case of deadlock. In this case, processes are waiting for each other but they are not blocked, they continuously try to catch resources and they cannot do it because of the scheduling or because they are waiting – it is active waiting – for each other. You can read about deadlock, livelock and starvation in [10], [71], [124], [144]. It is a hard task to design the system so that deadlocks, livelocks and starvations cannot be evolved.

If there are lots of processes working on a task together, then their work has to be synchronized. It can happen that a process has to wait for a partial result of an other one and they need to change data. There are several techniques to solve this problem. There are tools for solving the problem of waiting for other processes and others provide solutions to the safe communication and safe usage of resources (and some of them solve both problems).

In order to implement an effective synchronization, new structures are introduced. The most popular elements of these structures are semaphores

[30], [122]. Processes can be blocked on semaphores, and changing the value of a semaphore is done in atomic way (this is ensured by the hardware or the operating system). There are another structures that are linguistic elements and can be used for synchronizing processes or for providing a communication technique between tasks. There are for example monitors, channels and resources [18], [102], [144] for implementing synchronization and communication (they are not supported by processors). Monitor technique (was introduced by C. A. R. Hoare [63]) is one of the most popular linguistic structures of languages that support parallel programming (e.g. Java implemented monitors). An other popular technique is using channels. Channels can be used for communication between processes and by using synchronized channels, synchronization can be solved with them as well.

2.3 Object-orientedness and parallelism

Initially, object-orientation and parallelism originated and developed as separate and relatively independent areas [109]. It has been realized that parallelism is needed for the traditional object-oriented programming (OOP) paradigm, and that object-orientation can be usable in parallel programming. Primary OOP concepts such as objects, classes, inheritance, and dynamic typing can be used to model and simulate the real-world, so object-orientation was developed further which strives to analyse, design and implement computer applicatons through modelling of real-world objects. Many real-world objects perform concurrently with other objects, furthermore, they can work as objects of a distributed system, so the object-oriented paradigm needs to be extended with parallelism. It has been recognized, that users need parallel environments so programmers are to use parallel languages and language environments, therefore, it is needed to improve the areas of parallel programming. It is desired from parallel object-oriented programs to be as reusable as traditional serial object-oriented programs. For example, it would be desired that object-oriented languages provide better reuse of parallel software through the mechanisms of inheritance. Unfortunately, using inheritance in parallel programming is not so simple (see below). In this

thesis, we try to show an improvement that works in some cases of parallel object-oriented programs. However, several languages consider object-orientedness and parallelism as orthogonal concepts (e.g. C++). There are solutions that help the problem of parallelism by introducing declarative elements (e.g. Java). Parallel applications can be consistently and naturally developed through object-oriented analysis, design, and programming.

Despite of the progress of parallel object-oriented programming, there are many problems and difficulties that should be solved. Existing parallel OOP languages can often handle some important areas hard, e.g. inheritance, memory management and debugging [95].

Most languages are weak in providing inheritance for the synchronization code of parallel objects (see below in Section 2.3.2). The languages that permit inheritance with parallelism usually support only single-class inheritance and formulate many constraints. Furthermore, many languages cannot provide inheritance for objects which can be distributed in a network and which have methods that can run concurrently.

A large group of parallel OOP languages use C++ extensions for implementing parallelism. These languages are usually very complex and difficult to use. These languages usually handle parallelism not so efficiently. Interpretation-oriented languages (i.e. Java, Self, actor-based languages) usually have good parallel extensions but they provide not so good run-time efficiency. Fortunately, there have been great improvements in this field, so e.g. Java has a better run-time performance.

Nowadays, computing environments are becoming more and more heterogeneous. Users typically can use programs that have components on server machines and these components are desired to be reached from different clients that run on different platforms (e.g. PCs, PDAs, mobile phones). However, most OOP languages are implemented for homogeneous networks. There are solutions, e.g. CORBA [60] that solves the problem of distributed parallel applications, but it is very hard to use and does not provide the expected performance. Other solutions try to give a better performance by constraining the generality (e.g. DCOM [48], JRMI [102] or J2EE [103]) but these technologies reduce the portability of the applications (in the case of

J2EE efficiency causes an other problem because too complex logic is applied and it is not too easy to use).

Debugging parallel programs is very hard because their running is non-deterministic: it depends on the scheduler (if there are more processes than processors), running of which depends on the load of the system, among others. So if programmers see, that the program runs correctly without any faults, it does not mean that the next run will not throw an exception. So it is very important to provide tools which help programmers to make correct parallel object-oriented programs (e.g. by using a tool that helps specify and generate some parts of parallel programs). The other possibility is to provide a tool that can influence the scheduler so that all the execution paths can be tried during the debugging process by the programmer.

You can read more about object-oriented parallel programming in [109].

2.3.1 Shared objects

Shared objects are special objects, because their methods can be invoked in concurrent environments so methods of shared objects can run into multiple issues. This means that the methods of a shared object can use their attributes simultaneously, so modifying and reading these attributes have to be synchronized. If some method reaches a resource that has constraints for using it, then these methods has to be synchronized, too. Synchronization has to be specified so that programmers can write a correct program code easily by keeping the constraints of reaching the resources. If the methods of an object can be executed in parallel environments so that they are synchronized according to the specification of the resources they are reaching, then the object is thread safe. Shared objects are instances of shared classes. So, in fact, shared classes have to be designed to be thread safe.

Correct functioning of shared classes can be ensured also from outside. That is, objects that can invoke the methods of a shared object are designed so that critical methods of the shared objects can run only if it is safe (according to an outer specification). For example, if a method of a class modifies its attributes several times during its execution but the class is not thread

safe, then outer methods of the system that use an instance a of the shared class have to be written so that the mentioned method of a can run only if it is not executed by an other object (it can be ensured e.g. by using a global semaphore). It is a complex task and thread safe classes can be used more easily. There can be other problems. Assume that A is a thread safe class and it has a method m that writes to the output. In this case, if two objects, a_1 and a_2 are instantiated from A, then $a_1.m()$ and $a_2.m()$ can be executed parallely, so writing to the output is not safe. Because this is the problem of multiple objects, it is usually not considered as the problem of shared classes, and the safe execution of the functions of different instances is solved in the system – that is, thread safeness is defined in reference to an object of the class. But, of course, there are cases when the problem can be solved, e.g. if a class based language is used and there are class level – static – methods and attributes, then a static semaphore can ensure the desired behaviour. In other cases, with using a singleton pattern [25], [42], we can solve the problem of the global resources. Usually, good solutions can be made by designing a proper structure. For example, in Java, there is an object that writes to the standard output (System.out) and no other instances of its class can be instantiated.

You can read more about shared classes in [70] and [73].

2.3.2 Inheritance anomalies

As we mentioned before, problems can appear if we want to extend a class that has synchronization code. If instances of a class are used in a concurrent environment, then objects have synchronization code that is written in the mentioned class. If this class is extended, it is possible that methods have to be redefined because of an other redefined method. It can happen, if there are methods with synchronization calls, and the business logic of the methods do not change, but we have to change the synchronization of the methods, because of an other – perhaps new – part of the class.

If there are difficulties when reusing the synchronization code, then it is a case of inheritance anomalies. You can read about inheritance anomalies

in [16], [26], [92] and [127]. There are three major categories of inheritance anomalies.

- State partitioning anomaly.
- History-only sensitiveness of acceptable states.
- State modification anomaly.

State partitioning anomaly occurs in the acceptable set based schemes. State partitioning anomaly can be avoided with using guarded methods instead of acceptable sets. Unfortunately, writing of history-only sensitive methods is easy by using guarded methods. State modification anomaly can occure, if multiple inheritance is used (assuming the implementation of the synchronization with guarded methods). You can read more about inheritance anomalies and examples can be found in [96], and we will consider a special kind of state partitioning anomaly in Section 3.4.

There are solutions for the individual inheritance anomalies. For example, the history-only sensitiveness of acceptable states can be solved by using the algorithm of transition specifications. The using of synchronizers can solve the problem of the state partitioning anomaly. You can read about synchronizers and transition specifications in [17]. The solutions can be used only in special environments and for special cases. For example, the process model of the Eiffel language [128] defines a very good and very usable method to handle processes (special objects in Eiffel), but in concept of the efforts of the concurrency model of Eiffel, inheritance anomalies can occur by using that (but not so often).

Since there are lots of problems with handling inheritance in concurrent environments, Pierre America does not treat inheritance as a criteria of the object-orientedness in parallel environments (see [2]). It shows that although there are solutions for solving these problems, solutions are usually not complete and we have to take care of these problems when designing a new method for handling concurrency.

2.4 Specification of parallelism and synthesizing the synchronization code

There are several ways to specify parallelism. We can do it for example by using the classic predicate calculus, but by using predicate calculus we come up against several problems. For example, we cannot express that there has to be a time when a process is able to catch a resource. In fact, we can use only global and local invariants that cannot formulate liveness properties, but we usually need to be able to formulate such kind and similar limitations by specifying parallelism.

For specifying parallelism, we chose a natural extension of the classic predicate calculus, the branching time temporal logic [38]. In the following, we shortly summarize the bases of our method.

If we design a parallel system, we generally separate the synchronization code from the computation code [7]. We focus on the synchronization part of the code, and after the synchronization code was made, it is woven in the computation code. We will consider, how to specify the synchronization code of a parallel system, and how to synthesize it based on the specification. We have built our algorithm on the method of Attie and Emerson [7], so we give a short description of it.

2.4.1 Specification of parallelism by using temporal logics

There are many types of temporal logics [11], [37], etc. We chose a special kind of branching time temporal logics, the MPCTL* (Many-Processes CTL*) [7]. This specification language is an extension of the temporal logic CTL* (Computational Tree Logic) [38], which is a propositional branching time temporal logic. In CTL*, the following path quantifiers can be used: A (for all paths) or E (for some path). Linear-time modalities (that can be written after a path quantifier) are G (always), F (sometime), X_j (strong nexttime), Y_j (weak nexttime) and U (until). CTL* formulae are built up from atomic propositions, the Boolean operators $\wedge$, $\vee$, $\neg$, the path quantifiers

and the linear-time modalities.

CTL*

The syntax of CTL* is the following:

1. Every atomic proposition p is a state formula.
2. if f, g are state formulae, then so are $f \wedge g$, $\neg f$.
3. State formulae are also a path formulae.
4. if f, g are path formulae, then so are $f \wedge g$, $\neg f$.
5. if f, g are path formulae, then so are $X_j f$, fUg.
6. if f is a path formula, then Ef and Af are state formulae.

We consider the intuitive definition of the formulae mentioned above. Formula Ef means that there is a maximal path for which f holds. Formula Af means that f holds for every maximal path. Formula $X_j f$ means that the immediate successor state along the maximal path under consideration is reached by executing one step of process P_j, and formula f holds in that state. Formula fUg means that there is some state along the maximal path under consideration where g holds, and f holds at every state along this path up to at least the previous state.

The semantics of CTL* formulae can be defined formally with respect to a (K-process) structure $M = (S, R_{i_1}, \ldots, R_{i_K})$, where S is a countable set of states, each state is a mapping from the set of atomic propositions into $\{true, false\}$, and $R_i \subseteq S \times S$ is a binary relation on S giving the transitions of the sequential process with index i.

A *path* is a sequence of states $(s_1, s_2, \ldots)$ so that $\forall i : (s_i, s_{i+1}) \in R$ holds, and a *fullpath* is a maximal path. A fullpath may be finite or infinite. If $\pi = (s_1, s_2, \ldots)$ denotes a fullpath, then π^i is the suffix $(s_i, s_{i+1}, \ldots)$ of π where i is not greater than the length of π. $M, s_1 \models f$ means that f is *true* in structure M at state s_1, and respectively $M, \pi \models f$ means that f is *true* in structure M of fullpath π. $M, S \models f$ means $\forall s \in S : M, s \models f$ where S is the set of states. The semantics of $\models$ can be defined as follows.

1. $M, s_1 \models p$ iff $s_1(p) = true$
2. $M, s_1 \models f \wedge g$ iff $M, s_1 \models f$ and $M, s_1 \models g$
   $M, s_1 \models \neg f$ iff not$(M, s_1 \models f)$
3. $M, s_1 \models Ef$ iff there exists a fullpath $\pi = (s_1, s_2, \dots)$ in M such that $M, \pi \models f$
   $M, s_1 \models Af$ iff for every fullpath $\pi = (s_1, s_2, \dots)$ in M: $M, \pi \models f$
4. $M, \pi \models f$ iff $M, s_1 \models f$, where $\pi = (s_1, \dots)$
5. $M, \pi \models f \wedge g$ iff $M, \pi \models f$ and $M, \pi \models g$
   $M, \pi \models \neg f$ iff not$(M, \pi \models f)$
6. $M, \pi \models X_j f$ iff π^2 is defined and $(s_1, s_2) \in R_j$ and $M, \pi^2 \models f$
   $M, \pi \models fUg$ iff there exists $i \in [1 : |\pi|]$ such that $M, \pi^i \models g$ and for all $j \in [1 : (i-1)] : M, \pi^j \models f$

The logical disjunction, implication and equivalence can be introduced as usual. Furthermore, additional modalities as abbreviations can be introduced: $Y_j f$ is defined by $\neg X_j \neg f$, Ff can be defined by $trueUf$, and Gf means $\neg F \neg f$. Y_j is the "process indexed weak nexttime" modality, where $Y_j f$ intuitively means that if the immediate successor state along the maximal path exists, and is reached by executing one step of process P_j, then f holds in that state. F is the "eventually" modality, where Ff intuitively means that there is a state along the maximal path where f holds. G is the "always" modality, where Gf intuitively means that f holds at every state along the maximal path.

The complete description of CTL* can be find in [7].

The interconnection relation

The interconnection scheme between processes is given by the *interconnection relation I.*

Assume that we have K processes. Every process is indexed from the set $\{i_1, \dots, i_K\}$. We can define connections between these processes. $I \subseteq \{i_1, \dots, i_K\} \times \{i_1, \dots, i_K\}$, and $i \; I \; j$ iff processes i and j are interconnected.

I is symmetric, irreflexive, and total relation, thus every process is interconnected to at least one other process. If a process pair (i, j) is in I then i and j are connected. This means that running of i and j has to be synchronized. Only process pairs can be handled so if three processes are connected then three pairs have to be put in I, etc. If a process pair is not in I, the their running does not need to be synchronized.

MPCTL*

If we assume that every process is similar, we can specify that the synchronization of an arbitrary pair is valid for all pairs and every process has the same synchronization properties. These can be expressed in MPCTL*. Two processes are similar from the point of view of the synchronization, if their synchronization codes are equivalent, that is, their synchronization codes differ only in their indices.

An MPCTL* (Many-Process CTL*) formula consists of a spatial modality followed by a CTL* state formula over a "uniformly" indexed family $\mathcal{AP} = \{\mathcal{AP}_{i_1}, \ldots, \mathcal{AP}_{i_K}\}$ of atomic propositions. The propositions in $\mathcal{AP}_i$ are the same as those in $\mathcal{AP}_j$ except for their subscripts. A spatial modality is of the form $\bigwedge_i$ or $\bigwedge_{ij}$. $\bigwedge_i$ quantifies the process index i, which ranges over $\{i_1, \ldots, i_K\}$. $\bigwedge_{ij}$ quantifies the process indexes i, j, which range over the elements of $\{i_1, \ldots, i_K\}$ which are related by I. If the spatial modality is $\bigwedge_i$ then only atomic propositions in $\mathcal{AP}_i$, and if the spatial modality is $\bigwedge_{ij}$ then only atomic propositions in $\mathcal{AP}_i \cup \mathcal{AP}_j$ are allowed in the CTL* formula.

The definition of truth in structure M at state s of formula q is given by $M, s \models q$ iff $M, s \models q'$ where q' is the CTL* formula obtained from q by viewing q as an abbreviation and expanding it like

- $M, s \models \bigwedge_i f_i$ iff $\forall i \in \{i_1, \ldots, i_K\} : M, s \models f_i$
- $M, s \models \bigwedge_{ij} f_{ij}$ iff $\forall i, j \in I : M, s \models f_{ij}$

That is, MPCTL* formulae can express the properties of the processes or the features of the connected pairs. Because we assume that connected pairs are similar, the same atomic formulae can be used for every pairs.

You can find a complete description of MPCTL* (including CTL* and the interconnection relation) in [7].

2.4.2 Generating the abstract code from the MPCTL* specification

If there is an MPCTL* specification that describes the desired interactions of the processes, we can generate the synchronization code. The generated synchronization code is a finite non deterministic automaton. Only the synchronization of similar processes can be specified in MPCTL*.

Based on the specification, we can choose two interconnected processes and so the method of Emerson and Clarke (see [38]) can be used for generating the synchronization code of the two processes. The method of Emerson and Clarke is a tableau method [88], [89], [121], [143], which results in a finite non deterministic automaton. This automaton is the abstract program of the synchronization of the two processes. After that, the final synchronization code of an arbitrary process can be gained with using the algorithm of Attie and Emerson (see [7]) (because of the similarity of the processes), which is also an automaton.

We emphasize that this method can be used only if the processes are similar. If we want to extend this method for objects, this assumption cannot be used every time.

Chapter 3

The object-oriented synthesis method

In this Chapter, we summarize our results for designing and generating synchronization codes of parallel systems. We have built our method on the results of Attie and Emerson ([7]) by extending them to be able to be used in the case of object-oriented systems, while we exploit the features of the object-oriented programming. We published our earlier results in specifying and implementing the synchronization code of object-oriented systems in [54], [56] and [133]. We start with the simplest systems and progress to complex systems while considering interesting subproblems that have to be solved in order to achieve better performance. Before we present our solution, we take a short analysis to determine the environment we want to solve the task of generating the synchronization code in.

3.1 Analysis

In most cases, the synchronization code can be separated form the computation code. This means that objects execute their computation part and if they want to use a shared resource, they call the synchronization part of their code. We concentrate only for the synchronization part of the code (but we show later, how to connect it to the computation part automatically by

using a specification). As we will see below, we chose an approach, where synchronization steps are represented by state transitions, so if an object wants to do its task, it has to step in a corresponding state (or it will be blocked if it is not possible). Since state transitions represent the synchronization, and functions that implement them can be written separately, the synchronization code can always be separated from the computation code in our case. Note that the idea of separating the synchronization code from the computation code appears in Aspect-Oriented Programming [80], except that we wove the synchronization calls into the computation code.

If we design an object-oriented system, the best is to use a class based designing tool. If a well known tool is used for designing the computation code of the system, then it can be required to use a designing tool that is similar to the previous one, when designing the synchronization part of the code.

Our task is to extend an existing designing tool by adding the capability to make a parallel design which can be handled by our algorithm and suits well into the original tool.

Nowadays, UML is a very popular language (see [15], [135]) for designing object-oriented systems. From version 2.0 of UML there are diagrams with which synchronization of a system can be designed. Unfortunately, we have found several situations that cannot be described with those. We chose UML to extend with a new type of diagrams, furthermore, we extend existing UML elements for describing the structure of the synchronization code.

If we have a specification, we want to generate the synchronization code. We want to give a method that is able to generate concrete program code for different languages. There are several object-oriented languages but there can be significant differences between them. As it will be seen, we need to use a common access interface of the synchronization objects. So, it can help us, if interfaces [102] are supported by the used language. Instead of interfaces, (abstract) classes can be used if the language supports multiple inheritance [142].

We examined the languages that are popular nowadays. We used the next web site to gain some information about popularity of languages: in

[126], there is a list that shows the "popularity" of languages. This measurement is based on the statistics of the searching systems, where the search string is "+<language> programming". We experienced that object-oriented languages of the first ten elements support interfaces (e.g. Java, Basic, C#, Delphi), multiple inheritance (e.g. C++) or dynamic typing (Ruby) – we have not considered script languages, i.e. Python, Perl, PHP, because their object-oriented extensions are not so complete as it should be[1].

We have used class level functions in our solution, but if a language does not support class level functions, a globally used instance can be used instead of the static class (e.g. by using the pattern *Singleton* [25]).

According to these, we implement our examples in *Java* (see [44], [45], [46], [64], [84], [102]). If a concrete program code has to be written, we will write it in Java language but we will pay attention to write our program codes so that they can be transformed to an other language. We will give the implementation so that we examine, how it can be implemented in other languages with other properties but we focus only object-oriented programming languages that are widely used.

3.2 One-class system

In the case of a system that consists of entities (objects) that are instances of the same class, it is obvious that the state-sets of the entities are identical and the entities are similar. In a parallel object-oriented system objects work in parallel, while concurring for the resources. As in a one-class system objects are similar, the method of Attie and Emerson can be used if we are able to map this method into an object-oriented environment. However, there are many possible interconnection schemes of objects, which are handled by the interconnection relation I [7]. For instance, suppose that we have a class that has a size property, and we want only entities of size larger than $2m$ to be associated. However, objects can be created and destroyed dynamically and some properties that influence the connections of objects can be changed so

[1] Of course, the order of the languages changes continuously

we have to create a code that maintains the interconnection relation correctly.

It can be declared that in the case of a very simple system, when objects are instances of the same class, we only have to take care of the following problems:

- Objects can be created and destroyed, and the requirements of the interconnection relation have to be fulfilled. So the interconnection relation has to be dynamic.
- Because of the first condition, dynamic changes of the interconnection relation have to be specified and implemented in object-oriented environment.
- The structure of handling the interconnection relation has to be designed, and it has to be implemented.
- The global variables, which are generated by using the method of Emerson and Clarke (in [38]) have to be handled in object-oriented way.
- The state set of objects have to be defined and the generated synchronization code has to handle it correctly.
- Objects have to be created in their "initial" state, which allows any connections (because newly created objects have to be able to create their connections).

3.2.1 Designing the structure of handling the interconnection relation

As we mentioned in Section 3.1, we use UML for designing the structure of the synchronization part of systems. We will draft the desired behaviours of the synthesized synchronization and based on the expectations we make a model of the synchronization formulated in UML.

We consider I as a dynamic relation so we have to store it in the system. Furthermore, a procedure is needed that controls whether an object pair can get into set I. Typically, this procedure ($makeI$) is called with an object (as a parameter) when the object is created, and in that moment the object is

in some initial state that is permitted in any case – this is required from the specification –, so the object can get into I immediately. Procedure *makeI* also has to be called if an object wants to change its relations. In these cases *makeI* computes the new interconnection relation based on certain rules (see below). This shows that procedure *makeI* has to know the type of its parameter. Method *makeI* handles objects that have own synchronization codes. Synchronization code has to be implemented in the body of some functions. The name of these functions have to be known to generat the synchronization calls into the computation code. So we assume that every object that is in the set of objects that have to be synchronized, implements an interface. We called this interface *SynthesisObject*. So the type of the parameter of method *makeI* has to be *SynthesisObject*. The signature of the *makeI* procedure is something like the following (see [54]).

```
public static void makeI(SynthesisObject o);
```

We remark that in the case of other languages, where no interfaces exist, we can use (abstract) classes – e.g. C++ (see [123]) – and because of Section 3.1 we do not have to take care of other possibilities. But of course, it can be solved also in the case of languages that support no interfaces and no multiple inheritance. For example we can use pointers and we can make a solution where the first byte of the pointed structure describes the real static type of the parameter object and casting can be used in the body of the function *makeI*.

Function *makeI* creates new pairs in relation I and it removes the pairs that do not correspond to the specification of I (see below). But we need an other function, which is called *removeI* that removes all occurences of an object. This function is invoked before destroying objects. Function *removeI* does not check the specification of I, it only takes care of the object that is going to be destroyed.

In order to return a proper decision, where to place the function *makeI*, we have to make further examinations. I can be stored several ways but if we use objects, we usually store objects as references. For example when using Java (see [102]) or Delphi (see [19]), we can create new objects by storing only

pointers in our variables that are pointing to the real objects. Unfortunately, e.g. in C++ objects can also be referred as the objects themselves – by value. However, by using operator "&" we can get the address (the pointer) of an object, which is not stored as a pointer. As we experienced, every important language provides an operator that returns the address of a variable or forbids storing objects not as references. According to these, we can describe the pairs in I the most simple way by keeping reference-pairs, which point to entity-pairs, which will be synchronized.

We assume that this means that I stores no index pairs any more, but it contains reference pairs in our model. Similarly, in the MPCTL* specification of the synchronization we use object references instead of indices. That is, in notations $\bigwedge_i$ and $\bigwedge_{ij}$, i and j can be used as objects (i.e. i.method() can be used in the specification, etc. – but as it will be seen, we will limit of using methods). I can be implemented, e.g as a vector that stores records of the two pointers (see below).

In order that I can be maintained, when a new object is created, all participants of the system have to be known. So we have to maintain a list of the references of all the created objects. These values can be stored e.g. in a vector. We will refer for this vector as O in the future. Of course, it is enough to store this vector only once and functions *makeI* and *removeI* have to be stored also only once. We can introduce a new class that owns these attributes and functions as static members. Let this class be called *SharedObject* and let O and I be stored as static members of it (see *static Vectors* O and I in Figure 3.1). For reaching elements of I, two methods, *getICount* and *getI* are introduced.

The next question that arises is, what to store in relation I. As we mentioned, it is a good approach to store records of two pointers of *SynthesisObject*. A more general structure is if we describe these records via classes. In fact, records are special classes so we can use classes instead of records, and that is why we can extend these classes to provide more than simple object pairs. The class that represents object pairs is called *SynthesisObjectPair*.

Figure 3.1 shows the buildup of class *SharedObject*.

This approach for class *SharedObject* is appropriate because it contains

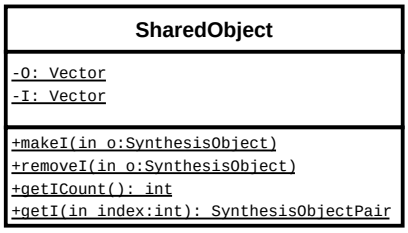

Figure 3.1: Class SharedObject.

only static members. If the used language supports no static members, an instance of this class can be created and this instance can be used instead of the static class. (Of course, in that case we have to take care of passing the instance to the other objects.)

In the relation I object pairs are stored. Because of this, we produced the *SynthesisObjectPair* class, instances of which store references of the corresponding objects, and instances of this class can be put into I. The method of Attie and Emerson (in [7]) generates an abstract program that uses global variables. These variables belong to a relation, because the method of Attie and Emerson uses the algorithm of Emerson and Clarke (see [38]) for synthesizing the synchronization code of process pairs, and this algorithm generates variables that are used only in the process pairs, furthermore, by means of the spatial operators these synchronization skeletons are transformed to the abstract program of an individual process, but the usage of the shared variables is kept.

So it is straightforward to store these variables in instances of class *SynthesisObjectPair*. Furthermore, a function is needed that shows whether an object belongs to this relation (*isInRelation*) and an other that returns the another element of the relation (*getRelationObject*). Figure 3.2 shows the *SynthesisObjectPair* class.

3.2.2 Specifying of the interconnection relation

We mentioned that relation I has to be dynamic in order to be able to handle creating and destroying objects. But if an object is created, it has to place

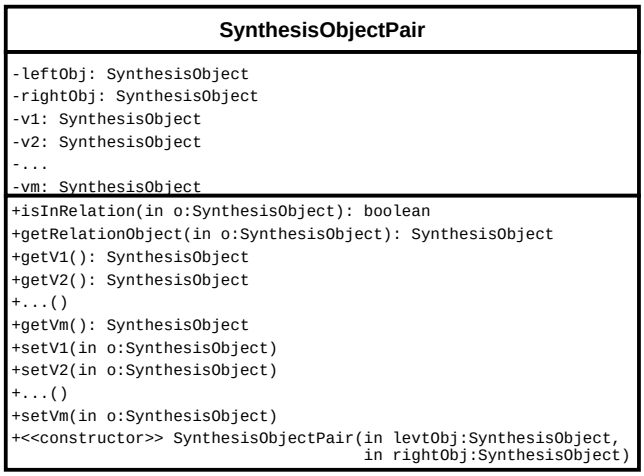

Figure 3.2: Class SynthesisObjectPair, which represents the relations between objects.V1, ..., Vm are the shared variables.

itself into I with certain other objects, and the rules of this procedure have to be specified. We wanted to choose a language that can describe all of the possibilities that can be specified. Let us examine what cases have to be described.

If an object is created, it has to be decided, which other objects are connected to it. But there can be also other cases when connections have to be created or removed. For example, assume that we want to synchronize two subsystems. The first one consists of objects with odd indices, the second one owns objects with even indices (we assume that every object has an *index* property). But it can happen that an object finishes its computation code and it can "help" the others in the second group, so it changes its index and it works together with the objects in the other object set. In such cases the existing connection set is changed.

We have to determine the elements that the interconnection can depend on. It is clear that attributes of an object can affect whether an object pair is in the interconnection relation. Of course, we cannot always use the attributes but in those cases public getter functions have to be invoked. Therefore, we can determine that public attributes and functions of classes

should be used for specifying the interconnection relation.

Because nothing else is needed for specifying the interconnection relation, it is enough to use classic predicate calculus for describing I where the variables are objects, the terms are the public attributes and functions of the classes (and the generally used operators of their types – e.g. operator "+" in the case of integers, etc.), the predicates are built up from terms and constants over the corresponding range of terms and operators "=", "$\neq$", ">", "$\geq$", "<", "$\leq$", and formulae are built up from predicates, the boolean operators and the quantifiers. An additional constraint is that exactly two free variables can be in a formula. If the value of the formula is true, then the currently substituted two objects have to be interconnected. The question is whether it is enough, and on the other hand, it can be asked if it is necessary (for example, it does not necessarily follow that using quantifiers is useful).

By using predicate calculus we can refer only to static expressions. For example, we can only require that two objects have to be connected if their states satisfy a condition in the actual time, but it cannot be expressed that two objects have to be connected if some conditions will be true for them. It is not a problem because if two objects have to work together at a moment, that cannot depend on values are valid in the future or in the past (or it can be stored in a variable, that can be used when specifying I). Detailed, the interconnection relation is dynamic, so it can be changed in every moment, so it is enough to watch only the variables in the present. Nevertheless, it has to be solved because of the dynamic behaviour of I, that no inconsistency is allowed via changing the interconnection relation.

Let us examine whether we can exploit all the features provided by the predicate calculus. Do we need quantifiers? Unfortunately we need them. Assuming that several classes have to be synchronized (see below) we may want to describe that an object is connected to an other object of an other type that has an *index* property and no other objects of this type exist that has a larger index (connecting to the end of a pipeline). So it is good if we can use everything that predicate calculus provides and it is enough for us to describe our properties.

Using spatial operators for specifying the interconnection relation

Referring to many-classes systems (see below) the interconnection relation is designed to be able to handle connections of objects with different types. Therefore, connections of objects of two different classes have to be defined too. We can use spatial operators (see [7], [54]) for describing these features, and in order to provide the possibility of describing different connection rules for different classes, we have to extend the spatial operators with some parameters. In fact, two parameters are needed that describe the classes over which the relation is defined. The syntax of writing these operators is $(c_1, c_2) \bigwedge_{ij}$. The intuitive meaning of these parameters is that if an object o_1 is instance of c_1 and an other object o_2 is instance of c_2 and they satisfy the formula, which follows the spatial operator, then they are connected.

$M, s \models (c_1, c_2) \bigwedge_{ij} f_{ij}$ iff
$\forall i, j \in O, i$ instanceof c_1, j instanceof $c_2 : M, s \models f_{ij} \rightarrow (i, j) \in I,$

where o *instanceof* c returns *true* iff o is an instance object of class c or o is an instance of a class that inherits – directly or indirectly – from c. There are exactly two free variables in f_{ij}: i and j.

In many cases we want to refer to all the objects of O in a common way, where $c_1 = c_2 =$ SynthesisObject. That is why we introduce a special notation:
$M, s \models (\uparrow, \uparrow) \bigwedge_{ij} f_{ij}$ iff $M, s \models (\text{SynthesisObject}, \text{SynthesisObject}) \bigwedge_{ij} f_{ij}$.

Let us see some examples for specifying the interconnection of objects. If we want to specify that every object is connected with each other, we can specify that with the following formula:

$$(\uparrow, \uparrow) \bigwedge_{ij} true.$$

If a class *IndexedObject* exists that implements the *SynthesisObject* interface and defines an attribute *index* with a getter function *getIndex()* and we want to specify that every object is connected to the only one that has the successor index, it can be written as (in this case the neighbours are connected):

$$(\text{IndexedObject}, \text{IndexedObject}) \bigwedge_{ij} j.getIndex() = i.getIndex() + 1.$$

If a class inherits form the other and a getter function is overridden then the dynamic type determines which getter is called at the implementation level. (In [54] we missed giving a specification method for the interconnection relation and *makeI* had to be written by hand. Now it can be specified and generated automatically.)

You can find an example for specifying a quite complex interconnection relation in Section 4.1.3.

3.2.3 Implementation of handling the interconnection relation

Implementing and calling the method *makeI*

Function *makeI* contains the implementation of the specification of the interconnection relation. The implementation works so that it examines the possible object pairs and checks whether they correspond to the specification. If two objects can be connected, *makeI* puts them in I. From the logical formulae of the specification of the interconnection relation an expression can be generated easily in the source code that can be evaluated by substituting the actual values of the attributes of objects. Handling of the existential and universal quantifiers can be solved so that checking iterates the object set, which is represented by vector O. We can put this generated expression in the body of a function:

```
boolean static checkI(SynthesisObject o1, SynthesisObject o1);
```

For example, in the case of the example where *IndexedObject*s were used and they were connected in a chain, function *checkI* can be generated as follows.

```
boolean static checkI(SynthesisObject o1, SynthesisObject o2) {
  boolean retVal = false;
```

```
  if ( o1 instanceof IndexedObject &&
       o2 instanceof IndexedObject )
    retVal = ((IndexedObject)o1).getIndex() ==
                 ((IndexedObject)o2).getIndex() + 1 ||
             ((IndexedObject)o1).getIndex() + 1 ==
                 ((IndexedObject)o2).getIndex();
  return retVal;
}
```

Function *makeI* looks for new object pairs by using *checkI*.

Because the interconnection relation is dynamic, we have to take care of satisfying constraints that are valid for I. Of course, if an object is created, it has to invoke the function *makeI* with itself as a parameter. Similarly, if an object is being destroyed, it calls *removeI* and passes itself as parameter. Because I depends on the attributes of objects, *makeI* has to be invoked if an attribute is changed. If *makeI* is invoked, it examines all the object pairs which contain its parameter, and checks whether the elements of the pairs satisfy the specification of I (by using *checkI*). If an examined object pair does not satisfy the conditions of I, *makeI* removes it from the interconnection relation. Besides, the function has to check all objects in O whether a new object pair can be found, term of which is the object that changes its property and if there are such pairs, it places them in I.

If an object a is created or one of its properties is changed, then all the connections of a have to be rechecked and *makeI* has to look for new connections, but if no quantifiers are used, the connections of other objects do not need to be examined, because their connections cannot depend on a (or they are checked because of a). If a new object pair is placed in I, it has to be checked that the new pair cannot be in I twice. If there are many objects and many connections, it is a very expensive task to examine the old connections (the full I has to be examined n times, where n is the number of the object, so it has an $O(n^3)$ behaviour – because the size of I is $O(n^2)$). So we optimized the handling of I as follows. If an object is newly created (it can be denoted e.g. with an attribute), *makeI* examines only the possible new pairs. If a property that cannot be found in the specificaiton of I is changed, then *makeI* is not invoked. In other cases, if an attribute of object

a is changed, then *makeI* examines the existing object pairs of a in I, and if an object pair (a, b) is examined, then it is denoted in an attribute of b – either the pair is deleted or is kept. After that, objects of O are examined and if the mentioned attribute is set for an object c, *makeI* only resets it, in other cases *makeI* checks, if the pair (a, c) has to be interconnected – and if the check returns true, they are put into I. It results an $O(n^2)$ algorithm.

Quantifiers may cause problems, e.g. if an object a changes its property so that (a, b) has to be put in I, it can happen that because of a quantifier another objects cannot be in the interconnection relation with b. Therefore, *makeI* works so that if quantifiers are used in the specification of I, then it recheckes all object pairs in I and examines all the possible new connection of the objects. By using the optimized algorithm above, it results an $O(n^3)$ behaviour. That is why this implementation is used only if there are quantifiers in the specification of I.

We remark that an other solution could be calling *makeI* before every synchronization step, but if no properties are changed then it is very expensive. As we expreienced, I depends on nearly static properties in several cases, so we chose to invoke *makeI* if a property that is relevant to it is changed.

Corresponding to the above, the generator algorithm works in the following way. During the synthesis algorithm, the set of the properties is built for every class, which contains all the properties that the interconnection relation depends on. After that the setter functions of these attributes are generated so that after setting the new value of the corresponding attributes they call the function *makeI*.

Unfortunately, we have to take care of an other problem. If the function *makeI* puts an object pair in I, it can cause inconsistency by violating the constraints of the synchronization specification (which contains MPCTL* formulae). We have two solutions.

- A property, that is relevant in the point of view of the interconnection relation, can be changed only if the object is in its initial state. In this case no check of the consistency is needed.

- If *makeI* runs for an object that is not in its initial state, then consistency has to be checked.

Let us see what we can do in the second case. There are two main types of temporal logic formulae of the synchronization specification. Formulae in the first type describe progress properties while instances of the second set formulate static properties. For example, formula $\bigwedge_{ij} AG(\neg(C_i \wedge C_j))$ describes that two objects cannot be in state C simultaneously (static property). In that case, we have to check all of these static properties. The synthesis algorithm collects formulae that contain constraints only for the actual state (and an AG operator is valid for them) and builds an expression that can be evaluated by using the current values of the objects. For example, in the case of the above example the following Java code can be generated:

```
boolean static checkConsistency(SynthesisObject o1,
    SynthesisObject o2) {
  return !(o1.getState() == C && o2.getState() == C);
}
```

where C is a constant value that represents a state. Of course, if this function returns *false*, then object, which calls *makeI*, has to be blocked until this function returns *true*. It can be implemented so that if function *checkConsistency* returns false, the object is blocked on a semaphore. If an object changes one of its properties that is important in the point of view of I, it wakes up the blocked objects that are connected to it (it can be solved similarly as it is described in Section 3.2.6 by the token capturing method). We remark that if *checkConsistency* is used, then this function cannot allow modifying I while the other object is in its condition evaluation state (see below).

If we choose the first implementation, then we only have to throw a new *exception* if an object tries to change one of its relevant properties while it is not in its initial state. An other approach is that we choose ostrich politics and do not take care of the consistency, that is, the generator assumes that *makeI* is called only in consistent states of the system and it is ensured by developers.

As it can be seen, the first solution is more faster but not so general than the second one. So if it is not needed to be able to change the value of a relevant attribute if objects are not in they initial state, then the first solution should be used, and the stronger second one can be chosen in complex cases. We assume that the first solution is used in the following, and if the second one has to be applied, we will emphasize it.

Implementing and calling *removeI*

As we mentioned, function *removeI* has to be called if an object executes its destructor. It is very good for languages like C++, where destructors are explicitly called. But there are languages like Java where a garbage collector works on freeing unused memory blocks. In these cases there is a function, which is called before garbage collector destroys the object. Unfortunately, garbage collector only destroys an object, when there are no references to it. In our case there are object references in vectors O and I, so garbage collector can never free memory blocks of synchronization objects. So we had to find a solution for it.

We introduce a new function, the *destroySynchronization* in the interface *SynthesisObject*, which executes function *removeI*. When an object in the computation code ends its working, a call for *destroySynchronization* is generated for its synchronization object (or if synchronization object is the same as the computation object, it "destroys" itself) – see Section 3.6 about connecting synchronization objects to computation objects.

The implementation of *removeI* deletes all object pairs from I, which contain its parameter. Besides, if quantifier are used in the specification of I, it has to recheck all the connections of I and examine all the possible new connections (see above).

Implementation of reading and writing I in parallel

The interconnection relation is used in parallel, so we have to synchronize of reaching I. There are methods that write I, and others can only read it. In fact methods *makeI* and *removeI* can write and method *getI* can read the

relation. Therefore, we have to solve the synchronization of these methods.

We use semaphores for implementing the abstract code. We decided for using semaphores because they are very general and they can be found in almost every popular environment. If they cannot be found in an environment, they usually can be implemented simply (for example, Appendix A shows a simple implementation of semaphores in Java). Furthermore, we implement objects so that they are communicating using shared variables. These shared variables are in fact attributes of objects but considering their concept in a parallel environment, this implementation in fact means communication through shared variables.

Of course, the case is not enabled when I is being changed by an object and I is being read by another object at the same time. This means that an object cannot evaluate transition conditions while another object is modifying I. Furthermore, writing I has to have priority against reading I – so new objects surely can build their new connections. To implement these restrictions let us introduce a counter *readCount* to count the objects that are reading I, and a counter *writeCount* to count the objects that are 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 *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 and *readWait* should be decremented.
- 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 *mutex*. Before every mentioned operation *mutex* should be let down and *mutex* should be lift up before an object is put to sleep or finishes its work. This solution is based on the implementation of the reader/writer problem which can be found in [73].

You can find the implementation of the mentioned functions (makeI, removeI and getI) in Appendix A.

3.2.4 Handling the synchronization of objects

We have to decide, where to store the synchronization code of an object that is in the synchronization set, that is, if the object has to be synchronized with some other ones. By using the method of Attie and Emerson [7], a finite non deterministic automaton (FNDA) is got as a result. Nodes of an FNDA represents states (these are the synchronization states, or simply states), and transitions between nodes can have conditions. It is clear that synchronization states can be represented by a set of the values of some attributes. If an attribute that can affect the synchronization state is changed, it has to be examined if a new synchronization state has to be set and running of the synchronization code is needed. Unfortunately, this approach makes the

synchronization specification too complex and the handling of the attributes becomes too difficult. So we decided to store synchronization states in a *state* attribute, which represents the actual synchronization states of objects. A *setState* method is introduced to change the value of the attribute, and function *getState* can be used to get the actual state. Developers have to define later, how an attribute affects the synchronization state (see Section 3.6).

Intuitively, when a designer prepares the state diagram of a class (we assume that the system is designed by using UML, so state diagram of classes are made), she determines computation state. She has to define computation states so that they have to be connected to the synchronization states and based on these – because the synchronization depends on the computation –, synchronization states can be set by using function *setState* (see Section 3.6). This means that function *setState* has to check, whether this state transition corresponds to the synchronization specification and it has to decide whether the new synchronization state is enabled and if it is not enabled, the object has to wait until the conditions enable the state transition. In fact, this means that the synchronization code is implemented in the method *setState* in our model.

The algorithm always has to know all the states in order to generate a proper code. So determining a state set is always necessary. The synthesis algorithm only generates the synchronization code as the function *setState*. But calling this function is implemented in the computation code: after some computation steps the synchronization code has to be called to continue the computation – the computation needs a proper synchronization state.

When using only one class, it can be assumed, that the state set can be produced from the synchronization specification, that is, objects are handled so that they can be in states that are written in the synchronization specification. Only valid states (and valid state transitions) can be handled in function *setState*. So if there is no state transition from the actual state to the new state, function *setState* throws a special exception, an *InvalidSynchronizationStateException*. In the case of many-classes systems (see below) we have to find an other solution for determining the state set of classes.

As we have stated above, every object implements *SynthesisObject* inter-

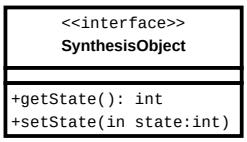

Figure 3.3: Interface SynthesisObject that has to be implemented by every object that has to be synchronized.

face. The UML model of *SyntesisObject* interface that can be seen in Figure 3.3.

Now we have all the classes, objects of which can be actors of a one-class system and can affect the synchronization. Figure 3.4 shows how they are connected. In the figure, *ConcreteObject* represents the only class, objects of which have to be synchronized. As it can be seen in the figure, that the objects of class *ConcreteObject* use class *SharedObject* for handling the interconnection relation. Every instance of ConcreteObject is in relation with the same SharedObject. It can be represented so that we use the class SharedObject as an object (that is SharedObject contains class level – static – functions) or by using the pattern *Singleton* only one instance is allowed to exist in the system and so every object uses that one. SharedObject administrate SynthesisObjects and it handles SynthesisObjectPairs. SynthesisObjectPairs are created to be stored in a collection of SharedObject so they are in *composition* relation. SynthesisObjectPair refers to exactly two SynthesisObject instances and they are in *aggregation* relation.

3.2.5 Specification of the synchronization

We use MPCTL* for specifying the synchronization of objects. The same method can be used that is introduced by Attie and Emerson for the case of similar processes [7]. When specifying the synchronization of objects, synchronization states can be used. As we mentioned before, spatial operators are indexed with objects, that is, if we write $\bigwedge_i$ then i denotes a reference to an object. This means that i.m() is a correct notation if class of i has a method "m". We have to define the synchronization of objects with type *Syn-*

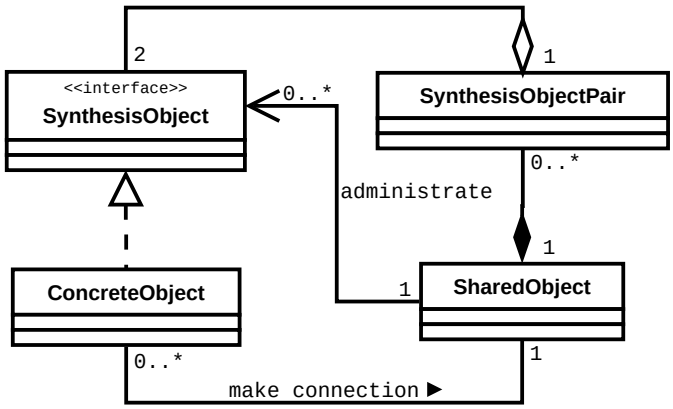

Figure 3.4: Connections of the synchronization classes.

thesisObject because of our assumption, that is, objects that index the spatial operators can be considered as *SynthesisObject*s. Therefore, we can use their methods and because we can use logical expressions in the specification, we define the atomic predicates as i.getState() $= X$, where i is a parameter of the interconnection relation and X is a state. Instead of i.getState() $= X$ we can write X_i in the specification in order to make specification more simple. After that, we can specify the synchronization of a one-class system by using MPCTL* formulae, similarly to [7].

Objects in more than one states

By using the approach above, objects can be in only one state. It is because function *setState* overrides the previous state. An alternative way exists to store states. We can use an array that represents the actual states of an object. In this case, function *setState* inserts a new element into this array and an other function, *unsetState* is required to remove an element from this array. Instead of function *getState* a function *isInState* can be introduced, which returns true if the object is in a given state, i.e. the state is in the state array of the object. Signatures of these functions can be expressed in Java as follows.

```
void setState(int state);
```

```
void unsetState(int state);
boolean isInState(int state);
```

In MPCTL* formulae *getState* is used in the following way: `o.getState() = X`, where `X` is a state constant. If we allow objects to be in more states, we can use the following notation to describe that object is in state X: `o.isInState(X)`.

So in this case objects can be in more states. Let us examine, whether we really need this extension! If an object is in two states, e.g. A and B, we can introduce a third one, e.g. C to express this fact with only one state. If an object is in three states, we can introduce a fourth one to denote that object owns all the three other states. So we can map this model into the model, where objects can own only one synchronization state. Because generating the body of *setState* is more difficult by using this model, we always require from the specification to formalize that objects can be only in one state, and if it is not satisfied, our generation method does not work.

3.2.6 Implementation of the synchronization

After specifying the synchronization in MPCTL* as it is described before, the method of Attie and Emerson can be used to generate the abstract program of the synchronization. In the implementation phase our task is to produce method *setState* of the class that implements *SynthesisObject*, which checks whether the state transition is possible in the actual circumstances.

To generate the body of *setState* based on an automaton is straightforward. We have to deal with the safe evaluation of conditions. We make the condition checker part on the basis of the conditions in the automaton and if a given condition is fulfilled then we execute the proper action part that is associated with the condition. The automaton 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, list of conditions and the list of the operations on the variables of the object pairs, corresponding to the conditions.

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. 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 status. That is why we have to assure that an object cannot 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 – the action is also finished, if the condition evaluation fails).

Some level of exclusion has to be provided in order to process the transitions, namely, no two objects can be in their condition evaluating or action executing phase at the same time. To solve this issue we use a token scheme like approach (see [13], [66], [97], [141]). 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 its connections. Hence, every element in I, which is an instances of *SynthesisObjectPair*, has a *token* attribute and two methods, *captureToken* and *releaseToken*. 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 objects. The return value of *captureToken* may be *true* or *false*. Value *true* indicates that the token is got successfully, and *false* indicates that the token is reserved. Method *captureToken* works in mutual exclusive mode, which is solved by using a semaphore.

In class *SharedObject* two methods, *captureTokens* and *releaseTokens* are implemented. These methods iterate through I and try to get/release every tokens that are related to an object, which is passed to these functions as a parameter. Unfortunately, there can be problems with capturing tokens. If a token cannot be captured because it is owned by an other object, then the caller must wait by using e.g. a semaphore. This solution would lead to an implementation where 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 tokens some time later after dropping. However, 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, and b from a. It fails and this process starts again with simple modifications. And so on. Starvation can appear similarly.

In order to avoid livelock we mention two methods. The first 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 cannot get a token then it releases all tokens it got and lifts up the semaphore. This semaphore practically should be placed in *SharedObject* because the obtaining of tokens is associated with I. In this case only one object is able to obtain tokens at a given time so livelock cannot take place. It can be seen that it is not the optimal method and it decreases the level of the parallelism. Furthermore, starvation can appear by using this solution.

The second method uses object priorities. If an object is created, it sets its priority to zero. If an object fails to get all the tokens it needs, it increases its priority and the priorities of all objects that are connected to it and have larger priorities than it. Increasing the value of the priority has to be implemented in atomic mode so a semaphore is used. For handling the priority, two public methods, *incPriority* and *getPriority* are introduced in the interface *SynthesisObject*. A variable *priority* is used for implementing these methods and a semaphore *semPriority* ensures that increasing the value of the attribute is safe. If an object wants to collect tokens, it denotes it: an *isCollectTokens* function is introduced in interface *SynthesisObject* and it has to be implemented so that it returns true if an object wants to get its tokens. If the object releases the tokens, function *isCollectTokens* should return false. If an object releases the tokens after its condition evaluation

and action execution phase, it sets its priority to zero. If two objects collect tokens and one of them (let it be object a) wants to get the common token, it examines the priority values. If the other object has a larger priority or it has the equal priority but it collected the common token yet, then a releases all the tokens but it remains in token collecting mode (that is *isCollectTokens* returns true and it does not set its priority to zero) and sleeps on a semaphore (this semaphore is an istance variable in the class that implements *SynthesisObject*). If a has larger or equal priority and the token is free then it allocates it. If a has larger priority but the token is not free, it notifies the other object that it has to waive its tokens and it sleeps on the mentioned semaphore but without releasing the collected tokens. Assuming that the other object may sleep while waiting for an other token, it wakes up the other object. If the other object already has all the tokens it needs, it first evaluates its condition then it releases its tokens and wakes up a. In fact, every call of *releaseTokens* wakes up the connected objects (we have to implement this method so that if the other object does not sleep, its waking up has to be a skip statement). If the other object has not collected all its tokens, it stops collecting the tokens in the next step, releases all the tokens it has (but it remains in collecting mode and keeps its priority) and wakes up a. It is clear that this method is appropriate for avoiding livelock as well as starvation.

We have to handle the case if *captureToken* fails to get a token. As we mentioned, the object has to sleep on a semaphore. We will consider it in details. We introduce a semaphore *tokenBlocker* with initial value of zero in all the direct descendants of interface *SynthesisObject*. Objects will be blocked on this variable. If an object wants to be blocked, it denotes it in a variable, named *tokenWait* (this variable is implemented in the same classes as *tokenBlocker*). If an object releases its tokens, it examines the values of *tokenWait* of objects that are connected to it and if it finds a true one, it wakes up the related objects. An other problem is that if an object fails to collect its tokens, it can happen that before it denotes that it wants to go to sleep, an other connected object that caused the fail, releases the token. For solving this problem we introduce an other variable, *tokenCollect* that

denotes whether an object is in its token capturing phase. This variable is set before beginning to capture tokens. Handling of *tokenWait* and *tokenCollect* is implemented in exclusive mode which is solved with a semaphore *tokenReleaser*. The algorithm is the following.

If an object begins to collect its tokens, it sets the value of *tokenCollect* to true in exclusive mode by using semaphore *tokenReleaser*. If it fails to capture all tokens it executes the following steps protected by semafore *tokenReleaser*: if the value of the variable *tokenCollect* is true then it sets *tokenWait* to true. It is done by using a function *setTokenWait* and if this function returns true, it executes function *doTokenWait* which forces the object to sleep on semaphore *tokenBlocker*. If an object releases its tokens, it calls the function *stopTokenWait*, which works in exclusive mode of each connected object. If the value of the variable *tokenWait* of a connected object is true, it is waked up. If the connected *tokenWait* is false, but *tokenCollect* is true, value of *tokenCollect* is set to false. In other cases there is nothing to do.

A process can evaluate the conditions of its state transitions only if it has captured all its tokens. If evaluating is succeeded then process executes the state transition and releases the tokens. This technique is similar to the "passing the baton" technique introduced by G. R. Andrews in [4]. The difference is that a "baton" belongs to a connected group and every group has an own baton. If an object cannot step into its destination state, it sleeps on a semaphore. If an other object executes a state change, wakes up the all the sleeping objects that are connected to it. It can be implemented so that objects use a semaphore with initial value of zero and a boolean variable that shows whether objects sleep on the semapore (we call this semaphore *blocker* and it is paced into every direct descendant of *SynthesisObject*). As it can be seen, in the case of capturing tokens we need an other variable that shows if the object is blocked in a semaphore. This variable is implemented in the same classes as *blocker* and is called *wait*. If an object evaluates its conditions then no other connected object can do it, so *captureTokens* works as a special semaphore: if it is succeeded then the next part of code is executed in exlcusive mode (of course it is concerned only for the connected objects). Because of the above, objects can set the value of variable *wait* before releas-

ing tokens safely. After that, objects release their tokens and are blocked on semaphore *blocker*. If an other object executes a state change, it releases its tokens and calls the function *stopWait* of all connected objects. This function works in exclusive mode and examines values of the *wait* variables and if it finds a true one, it wakes up the related object.

An other problem is that objects can remove themselves from the interconnection relation while others are examining their condition of transactions or capturing tokens. The solution for this problem is that if an object wants to remove itself from the interconnection relation (that is, it is going to be destroyed) it captures all the tokens it needs. Therefore, function *destroySynchronization* has to be written so that it captures the tokens, and then it removes the object from I. Releasing tokens is not needed because it removes all the object pairs tokens of which it can own. So it cannot change I while others are in their evaluation part. Because objects also can collect tokens while an other is changing I (e.g. removes elements from I), method *captureToken* has to handle the cases if objects from I are destroyed before examining their values. Furthermore, because an object can block others by keeping the tokens, if an object removes an object pair from I, it has to wake up the connected one.

Similarly, if an object (let it be a) is created and it puts *SynthesisObjectPair*s into I, but one of the related object (let it be b) already collected all its tokens, then it can cause problems if a begins to collect its tokens. In this case a can get the common token of a and b and so two objects can be in their condition evaluation sections. To avoid this problem, if an object is created, it creates its *SynthesisObjectPair* entities so that it sets the value of the tokens to the connected objects if they already own tokens (it can be checked safely because I is locked if a new object pair is being put in it).

It can be seen that implementing of the synchronization can be made by using constant codes. For example, the methods that are related to tokenizing are the same in the case of all systems and handling unsuccessful state transitions also can be implemented with constant parts in function *setState*.

Based on the written above, interface *SynthesisObject* has to be extended with the following functions:

```
public void stopWait();
public boolean getTokenCollect();
public void setTokenCollect(boolean tokenCollect);
public boolean startTokenWait();
public void doTokenWait();
public void stopTokenWait();
public boolean isCollectTokens();
public void incPriority();
public int getPriority();
```

According to these an implementation of this synchronization solution can be seen in Appendix A.

3.3 Many-classes system – without inheritance

In the case of a system that contains many classes the only change compared to the one-class case is that the state-set of the individual objects may differ. For example, it is natural that an object that represents some kind of animal can be in a "hungry" state, while a plant has no such state. So we have to find a method this problem can be handled with.

3.3.1 Separate the synchronization code from the computational code on the level of classes

Because the computational code and the synchronization code can be separated, it is clear, that there are cases when the system consists of several classes, and some classes are equivalent in the point of view of the synchronization. For example, if a simulation environment is given where there are users who use a database, and jobs (programs) can use the same database, then users belong to class *User*, jobs belong to class *Program* but they may have the same synchronization code: if any object writes the database, no other object can read or write it, but parallel reading is allowed.

So, in such cases the synchronization code should be implemented only once. But the synchronization code is placed in function *setState*. So both classes have to implement the *SynthesisObject* interface by implementing the

setState function. In this case, the same synchronization code would be stored twice.

A better solution is if a common ancestor implements the *SynthesisObject* interface and both class (*User* and *Program*) extends it. The problem is, that it may happen that class *User* has to extend a class that implements some code that is specific for humans and class *Program* has to extend an other program-specific class. If the target language does not support multiple inheritance, then the classes cannot extend the common ancestor.

Furthermore, a solution like the above can make the plan and the source code too difficult and hard to understand. A better solution is if we separate the synchronization code from the computation code on the level of classes. This can be done so that we prepare the plan of the system without synchronization and based on this we determine different types of synchronization, and design the sturcture of it. In our case, there is only one type of synchronization: reading and writing a database. After that, we connect the synchronization part to the computational part so that classes that are containing the computation code refer to the corresponding classes with their synchronization code. Figure 3.5 shows this separation in the case of the above example.

Based on this, we can consider only a part of the class diagram, the part that contains only classes that represent synchronization codes. Of course, such step is not always needed but in many cases we can simplify our system with this, e.g. to a one-class system, as the example above shows. In the future, we consider only the classes from the class set of the synchronization part of the system. We remark that only attributes and methods of the synchronization part can be used by specifying I.

In these cases objects that belong to the computation part call the function *setState*. How can they decide which synchronization states belong to their actual state? On implementation level, the computational code calls the synchronization code. There are two choices. The first is, that we assume that developers of the computation code exactly know, which synchronization states belong to the individual computation operations and we entrust calling function *setState* to them. The second – and of course better – choice

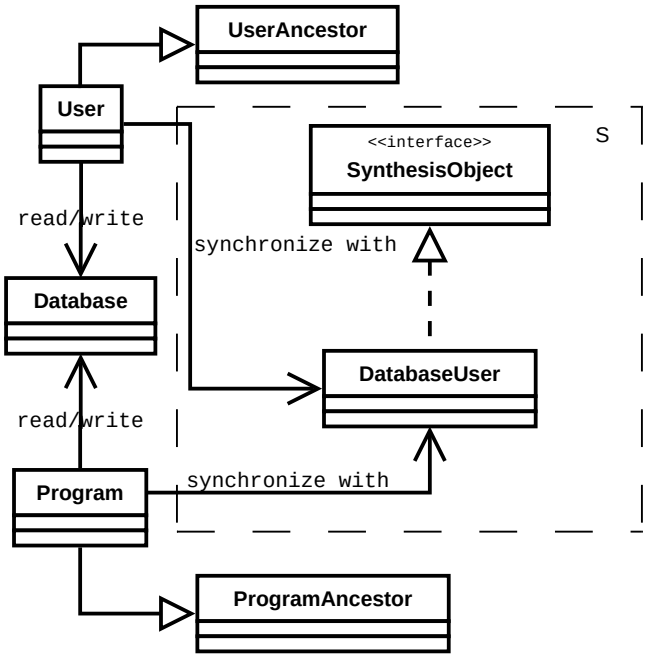

Figure 3.5: Separating the synchronization part from the computational part. The synchronization part is labelled with S.

is to connect the synchronization calls with the computation code at specification level. So we have to specify this connection. As we use UML for planning a system, we will extend UML so that we can do this. We will consider this topic later (see Section 3.6).

3.3.2 Handling more synchronization classes

In the case of systems that consist of objects of several classes we cannot use the algorithm of Attie and Emerson because objects are not similar. We can solve the problem of the many-classes systems by simply taking the union of the state sets of all classes, and then consider the synchronization code above this set of states.

Let us examine what problems it can cause. We want to avoid the problem of the state explosion. By applying the method of Emerson and Clarke, the state explosion problem can arise, that is, the synchronization code of a

system is going to be exponentially complex with the number of processes. By mapping the algorithm of Attie and Emerson we can solve this problem. What happens if we union the state set of classes? We get the largest number of states if the state sets of the classes are disjunct in couples. We get the following upper bound for the cardinality of the system's state set:

$$|\text{system's state set}| \leq \sum_{i \in CS} |i\text{'s state set}|,$$

where CS is the set of classes objects of which have to be synchronized.

In most cases, $|CS| << |\{\text{objects that have to be synchronized}\}|$, but the real power of the state union is that the algorithm of the one-class system can be used, so increasing the number of objects does not result increasing the complexity of the synthesis. So instead of a synchronization problem that is growing exponentially in the number of the objects we got problem that is $O(n^2)$ in the number of classes.

Unfortunately, this solution has many disadvantages as well. For example, if we have many classes, we introduce many new states for objects (because objects get the full state set), but the objects cannot get into their "new" states. It means that after the synthesis method lots of additional post-operations have to be executed to throw off the undesirable states form the synchronization code of classes. Furthermore, by specifying the synchronization, we have to deal with many types. For example if objects with type T_1 can step only into state B from state A, but objects with type T_2 can step only into state C, then it has to be denoted in the specification that both steps are allowed (after the synthesis we eliminate the inaccessible states from the program of the classes). This makes the specification incomprehensible (developers can maintain the specification harder).

As we mentioned before, this way the classes can refer to states of another classes which will never be taken but are important in terms of the synchronization. (So these states can be deleted from the nodes of the automaton but they have to be kept on the edges – see Section 3.4.3.)

Considering that the state union significantly increases the number of states, we should try to simplify our model during the design phase. For

example, suppose that we are describing a system of passenger and vehicle traffic. Suppose that no two passengers or vehicles can be in the same place. In this case "being in a given place" is critical section for objects of both classes, so it can be handled as "critical section" state in the instances of both classes. Thus, in some cases there are states that can be merged. It can be thougth that in the example above the two critical states cannot be merged because it can happen that in the case of class Passenger objects step into state critical from a state A, while in the case of class Vehicle, objects reach the critical state from a state B. In fact, it cannot lead to problems because we describe the rules of stepping into the critical state from either state A or state B (in the MPCTL* specification). It leads a FNDA as an abstract program where there are two edges pointing to the critical state: one from state A and an other from state B. Because a passenger cannot be in state B and a vehicle cannot be in state A, the edge from state A refers to class Passenger, while the other belongs to the program code of class Vehicle. So all we have to do is to split this automaton into two parts, the part of the first class, and the part of the second.

3.3.3 Storing enabled states

In the case of more classes we make a synchronization specification, which is referred to all classes, because objects can affect each other. So we use all the states in the specification. But objects cannot have all the states, and they can step only into states that are characterized to their classes. So enabled states have to be stored in the classes. After synchronization, we can split the synchronization code to parts according to the classes, which is based on these state set definitions (see Section 3.4.3). This means that in specification level we have to describe the enabled state set for every synchronization class. Because users may want to call the synchronization function (*setState*) by hand, enabled states have to be specified somewhere explicitly. We decided to specify enabled states in the class diagrams. Enabled states are constant integer values and are marked with a stereotype "«state»".

3.3.4 Merging states in the joint set

According to the findings above, we would like to find an algorithm that searches for states that can be merged. Unfortunately, an MPCTL* specification is not enough to merge states but we have no more information about states. If we use only a temporal logic specification, we can only give an algorithm that has a good chance to make mistakes.

Unfortunately, we have not found a usable algorithm for merging states, developing a good solution is a part of our research work in the future. So our algorithm only merges some special states, e.g. they match to the pattern of critical states of the mutual exclusion, etc. Furthermore, users are allowed to merge states if they know that two states describes the same.

3.3.5 Specification and implementation

Specifying a system that is composed of many similar objects is more difficult than giving a simple temporal logic specification, since we have to handle all issues above.

The specification can be produced by the following steps:

1. Design the static structure of the system by using UML class diagrams (separate the synchronization code if needed and connect the structure of the synchronization part to the computation part – with associations).
2. Perform the combining of the state-sets of the classes. We can assume that the state sets of the classes are disjunct (it can be easily achieved). Thus combining the state-sets is actually producing the disjunct union of the individual state-sets. (Of course we only deal with the synchronization part of the class set.)
3. Reduce the sets by merging the states of different types which formulate identical constraints for the object.
4. Connect the computation code to the synchronization code by using UML state diagrams (We will consider this solution later.)

5. Give the temporal logic specification in MPCTL* as in [7].
6. Execute automatic state merging based on the MPCTL* specification (in some special cases).
7. Give the specification of the interconnection relation.

After specifying the system, the synthesis algorithm is called, which generates the concrete program code based on that. We remark that generating the skeleton of the computation code can be done by using the code generator of a CASE tool. Connecting the structure of the computation part to the synchronization part results that the generator of the CASE tool creates the references to the synchronization classes in the computational part and generates instantiate statements.

Connecting the synchronization code to the computation code results invoking the function *setState* of the synchronization object in the computation code. We discuss it later in Section 3.6. Connection of the synchronization code means of course calling also functions *makeI* and *removeI* in the proper places (see Section 3.2.3).

After the abstract program of the synchronization was generated, the synchronization skeleton of the classes can be generated by projecting the skeleton of the system to the proper set of states (that is, different automatons are given for different classes by deleting the unreachable states). After that, every *setState* can be implemented as it is described in Section 3.2.6.

You can see an example for specifying the synchronization and the resulted synthesized code in Appendix A.

3.4 Handling inheritance

If a class is a descendant of an other data type, it inherits the states of its ancestor. In this case the state-sets are not disjunct. This raises certain problems because it is possible that states might have to be handled other way in the descendant.

As an example, consider the following situation. Assume that we want to make the simulation of a surgery. In this simulation there are patients, who

go to a doctor to make themselves examined. Of course, if a doctor examines a patient, another patients cannot be in the examination room, they have to wait in the waiting room. So we can define the synchronization of the system this way. In this case, we have the following states: normal (N), in the waiting room (W) and in the examination room (E). In fact, this is the case of the mutual exclusion problem and its solution can be found in [7].

We may write the simulation of a system, where there can be severe diseased patients and we want to specify that a severe diseased can go to the examination room immediately, which in fact means that they do not have to be synchronized (but if a severe diseased is in the examination room, then other, normal patients cannot step in). In the simulation world we can think of severe diseased patients so that they are marked as severe diseased in the ambulance. Creating of normal patients and severe diseased patients can be imagined in the world of the objects so that two classes can be instantiated: Patient and Severe, and both of them get a human object (these human objects represent the people who can be taken ill, and objects that represent their illness can be nested into humans). In this case, we cannot define the specification because state E has to be handled differently in the case of severe diseased and normal patients.

A possible solution for this problem is the following: the synchronization class of the severe diseased patients inherits from the synchronization class of the normal patients and a new state is introduced in the descendant: "severe in the examination room" (S). After that, there are the following states in the system (after unioning the states): N, W, E and S. (That is, states N and W are common states – they can be handled like merged states of two different classes.) We can specify the system so that we add some additional MPCTL* formulae and we formulate the requirements of handling the new state (S). It would be desired if the synchronization code of the ancestor class was not regenerated. Furthermore, we want to use the synchronization code of the ascendant in the descendant if it is possible, that is, if not a state transition to state S has to be executed.

We can solve this problem easily by hand: we do not need to introduce the new state S and we can override the function *setState* so that if its goal

state is E, then the state transition is allowed by all humans (that is, it does not block the caller of function *setState*). In other cases the implementation of the ancestor can be used. Of course this is not possible automatically. We would like to find a solution that the synchronization code can be generated with and the ancestor do not have to be regenerated. We will examine this problem in details in Section 3.4.3.

3.4.1 The generated synchronization code

If we specify a new system, then the synchronization code of the full system is generated and after the generation we split it according to the used classes. This is made by the state sets of classes. In the case of a class that inherits from an other, separation is special: we generate the function *setState* of the descandant so that if both the start state and the destination state can be found in the ancestor, then the code of the ancestor will be executed (it can be done because of the construction of the synchronization code of the individual classes), else new code is written. After executing the new code, we have to set the new state. Classes that implement directly the interface *SynthesisObject*, introduce a *state* variable. These are the direct child classes. As it is described above, classes that extend these direct children have to be able to set variable *state*. So direct child classes declare a function *setStateVariable* which assigns value to variable *state* and these functions can be used in the indirect child classes (so it has to be *protected*).

There are cases when the descendant does not use all the states of the ascendant. In these cases it has to be denoted in the class diagram by writing the mentioned state with a special stereotype "«exclude»". (For example, in the case of the example of the surgery, state E has to be marked as «exclude» in class Severe.)

If we extend a class of an existing system, then it is desired to reuse the synchronization code of the ascendant and generate only the synchronization code of the new class. In fact, the synchronization code of the ascendant class has to be regenerated in many cases because synchronization describes behaviours of objects and the behavioural properties can depend on other

objects. So it is clear that if an other type is created in the system, the old types have to handle this, so their synchronization code may change. We will examine, what we can do to avoid this problem (see Section 3.4.4).

Regenerating the code of the ascendant classes can be done without coding by hand but it can cause some problems. Assume, that we have a system, and we use a class in the system, which has synchronization code, and the structure of synchronization code is not separated from the structure of the computation code. Furthermore, we do not own the source code of this class. Besides, we want to extend this class so that its synchronization code has to be changed. Then a very poor solution can be applied: we regenerate its synchronization, introduce a new class that extends the previous one and we use the new class in our system (of course, the descendant class contains the new synchronization code). It is a good solution if no ancestor entities are instantiated. But if the original class is used frequently in the program, therefore its modification would be difficult, then code refactoring (see [137], [28], [69], [32], [111]) is needed, which may not be the best solution.

3.4.2 Dynamic synchronization behaviour

By using this kind of separation that is mentioned above and is demonstrated in Figure 3.5, dynamic synchronization behaviour can be assigned to objects that belong to the computation code. It can be done so that an object instantiates a descendant of its synchronization class instead of its static synchronization class. In this case, using function *setState* results of running the implementation of the child class because of the late binding.

This dynamism can cause problems, namely state sets can differ in the case of the ascendant and descendant classes (see the example in 3.4.3). So the computation code has to be connected to the synchronization code correctly to execute the correspondent calls of *setState*. We will consider later, how to assign the corresponding part of code in a secure way so that it can handle such cases.

Consider the example of a surgery above. There are two synchronization classes, patient and severe. The computation code can be written so that it

uses humans, and humans can take the role of a normal or a severe diseased patient. It can be written so that humans instantiate a patient if they are ill and severe if they are severely ill.

If an object instantiates a new synchronization object, it takes a new behaviour and drops the old one. This means that handling vector I is needed so it can be expensive. A solution can be if objects do not remove their pairs from I, they only mark themselves as not working objects. If a connected object examines their states and realizes that they are not working objects, it removes the object pairs from I (a new function has be introduced in *SharedObject*). This solution is not a good choice if there are objects that change their synchronization behaviours frequently. In these cases I grows too large. We do not consider the case, when an object might have more synchronization entities. So every object can own a synchronization object and if a new synchronization behaviour is taken, the old one is dropped. (An example can be seen for it in Appendix A, where class *Human* can own two types of synchronization behaviours and it generates a random number to choose one – it goes to the surgery as either a normal patient or a severely ill.)

3.4.3 Extending a synchronization class – Example

We consider the example of a surgery (see above). We give a formal solution for the synchronization problem, but we omit the structural design – you can see a full example of generating the synchronization code in Appendix A. There are normal patients that can be in normal state (N), in the waiting room (W) or in the examination room (E), and the ordered sequence of the states must be $N \rightarrow W \rightarrow E$. The severely ills can be in states N, W and in a new state, severe diseased in the examination room (S), and they can always step into S from W (and the ordered sequence of their states is $N \rightarrow W \rightarrow S$). Furthermore, if a severe diseased patient is in the examination room, normal patients cannot step in.

The synchronization code can be implemented by using two classes: *Patient* and *Severe*, which extends class *Patient*. Merging the sets of classes

means that the states of entity O_i are from set $\{N, W, E, S\}$ (the appropriate atomic propositions are N_i, W_i, E_i, S_i – N_i means that object with reference i is in state N).

In our example, every object is connected to all the others because patients affect each other. So the specification of I is the following:

$$(\uparrow, \uparrow) \bigwedge_{ij} true.$$

The corresponding part of the generated function *makeI* in Java is the following:

```
private static java.util.Vector<SynthesisObject> I =
  new java.util.Vector<SynthesisObject>();
private static java.util.Vector<SynthesisObject> O =
  new java.util.Vector<SynthesisObject>();
private static boolean checkI(SynthesisObject o1,
    SynthesisObject o2) {
  return true;
}
public static void makeI(SynthesisObject o) {
  ...
  for ( int i = 0; i < O.size(); i++ )
    if ( checkI(o, O.get(i)) )
      I.add(new SynthesisObjectPair(o, O.get(i)));
  O.add(o);
  ...
}
```

We emphasize that the functions above do not contain the full code because vector I has to be accessed in exclusive mode! We have considered these constraints in Section 3.2.3. We remark, that as function *checkI* returns true for every object pairs, *makeI* only has to search for new connections of the newly created objects, so *makeI* can be implemented in an optimized way (see above and Secton 3.2.3). The state sets are the following:

- State set of class *Patient* = {N, W, E}
- State set of class *Severe* = {N, W, S}

The temporal logic specification of the system is the following:

1. Initial state (Every object is initially in its normal state. The statement is trivially true for the objects that are not entered into the system yet, because they are not in O):

$$\bigwedge_i N_i$$

2. It is always the case that any move O_i makes from its normal state is into its "in waiting room" state, and such a move is always possible:

$$\bigwedge_i AG(N_i \Rightarrow (AY_i W_i \wedge EX_i W_i))$$

3. It is always the case that any move O_i makes from its "in the waiting room" state is into its "in the examination room" or "severe the in examination room" state:

$$\bigwedge_i AG(W_i \Rightarrow (AY_i(E_i \vee S_i)))$$

4. It is always the case that any move O_i makes from its "in the examination room" state is into its normal state, and such a move is always possible:

$$\bigwedge_i AG(E_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

5. It is always the case that any move O_i makes from its "severe in the examination room" state is into its normal state, and such a move is always possible:

$$\bigwedge_i AG(S_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

6. A move O_i makes from its "in the waiting room" state is into its "severe in the examination room" state is always possible and O_i does not starve:

$$\bigwedge_i AG(W_i \Rightarrow (EX_i S_i \vee AFE_i))$$

7. O_i is always in exactly one of the states N_i, W_i, E_i or S_i:

$$\bigwedge_i AG(N_i \equiv \neg(W_i \vee E_i \vee S_i)) \quad \bigwedge_i AG(E_i \equiv \neg(N_i \vee W_i \vee S_i))$$
$$\bigwedge_i AG(W_i \equiv \neg(N_i \vee E_i \vee S_i)) \quad \bigwedge_i AG(S_i \equiv \neg(N_i \vee W_i \vee E_i))$$

8. A transition by an object cannot cause a transition by another:

$$\bigwedge_{ij} AG((N_i \Rightarrow AY_j N_i) \wedge (N_j \Rightarrow AY_i N_j)) \quad \bigwedge_{ij} AG((E_i \Rightarrow AY_j E_i) \wedge (E_j \Rightarrow AY_i E_j))$$
$$\bigwedge_{ij} AG((W_i \Rightarrow AY_j W_i) \wedge (W_j \Rightarrow AY_i W_j)) \quad \bigwedge_{ij} AG((S_i \Rightarrow AY_j S_i) \wedge (S_j \Rightarrow AY_i S_j))$$

9. No two objects access "in the waiting room" state together:

$$\bigwedge_{ij} AG(\neg(E_i \wedge E_j))$$

10. A move O_i makes from its "in the waiting room" state is into its "in the examination room" state is not allowed if a connected object is in state "severe in examination room":

$$\bigwedge_{ij} AG((W_i \wedge S_j) \Rightarrow \neg EX_i E_i)$$

The synthesized synchronization skeleton of the system that consists of patient and severe diseased objects is shown in Figure 3.6. Notation X_i means that object with reference i is in state X, namely `i.getState() = X`. Notation $x_{ij} := j$ means `p.setV1(j)` (only one variable is used), where $p \in I$ and `p.isInRelation(i) = true` and `p.getRelationObject(i) = j`. Similarly, $x_{ij} = i$ means `p.getV1() = i`. Transitions are composed of *enablers*, which are connected by two operators, $\oplus$ and $\otimes$. An enabler is composed of two parts: a boolean part and an action part separated by the operator $\rightarrow$. The boolean part is a predicate calculus expression over the states while the action part is a program code (value assignments) has to be executed if the condition part is fulfilled (the condition evaluation and the action execution have to be done together, in mutual exclusive mode). Notation $\oplus$ means "or" operation, that is a transition is enabled if the condition part of an enabler separated by the operator $\oplus$ is fulfilled. Notation $\otimes$ means "and" operation, that is, a transition is enabled iff every part of the transition connected by the operator $\otimes$ enables the transition.

A normal patient can never be in state S and a severe diseased never can be in state E so these states may be removed from the synchronization skeletons of the appropriate objects. The result is shown in Figure 3.7.

If class *Severe* does not exist in the system, the synchronization of class *Patient* differs because it currently uses state S, but without *Severe* this state is not used. So the synchronization of *Patient* has to be regenerated, which is a special inheritance anomaly.

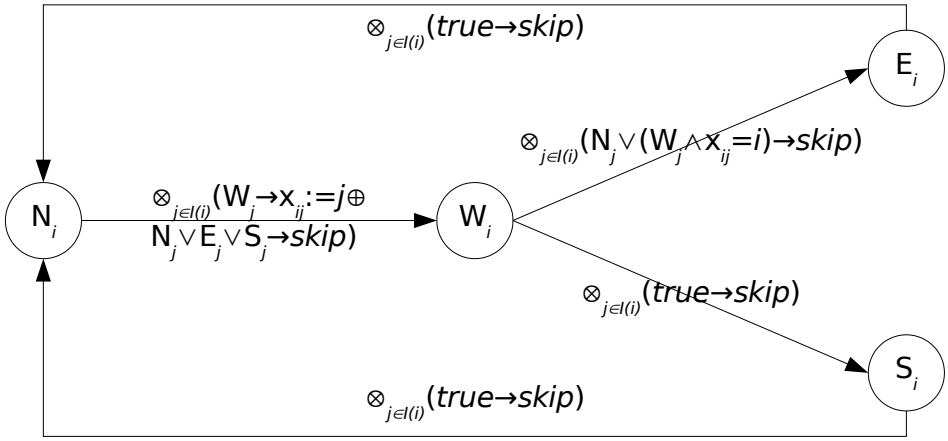

Figure 3.6: Synchronization skeleton of the example surgery system.

3.4.4 Forcing the synchronization code not to be regenerated

There are cases when the synchronization of the ancestor class has to be regenerated because it has to handle states that are introduced in the descendant class. We examine, what we can do to avoid this kind of problems. Considering the example above, it can be seen that the synchronization code of *Patient* has to be modified if class *Severe* is introduced. This problem can be solved with a simple trick: notice, that all states are used in the condition of the state transition $N \rightarrow W$. So implementing this code with an "else" statement, the synchronization code of *Patient* is the same if *Severe* is used and if it is not. But there are more complex cases. We can modify the specification so that normal patients can step into the examination room if a severe diseased is in the room because severely ill patients are examined by an other doctor. In this case the condition of the $W \rightarrow E$ would be: $\otimes_{j \in I(i)}(N_j \vee S_j \vee (W_j \wedge x_{ij} = i) \rightarrow skip)$. Assume that the synchronization of a system which consists of only normal patients is synthesized and we would like to derive the class *Severe* from the class *Patient*. In this case the synchronization of class *Patient* would be regenerated. In fact, it means that a special case of the state partitioning anomaly occurs (see [26] and Section 2.3.2).

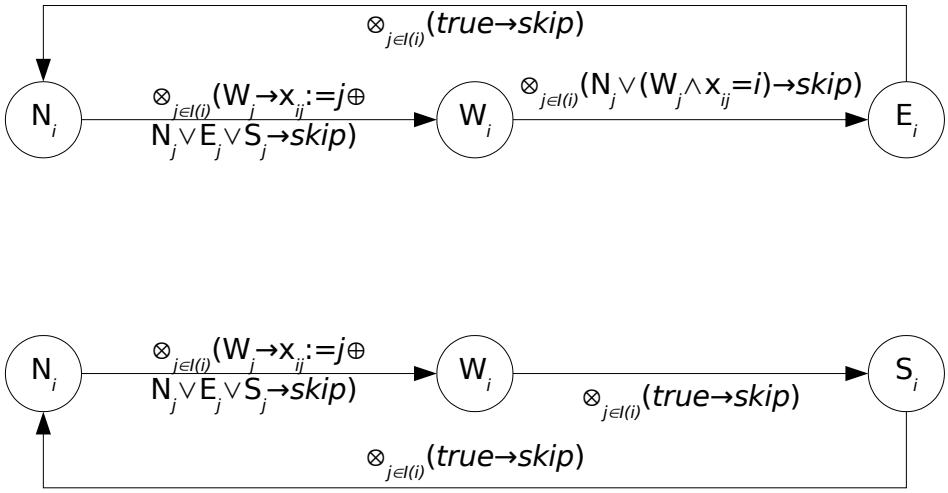

Figure 3.7: Synchronization skeleton of class *Normal* (above) and class *Severe* (below).

In order to avoid such problems we introduced the set of equivalent states. Recognize that state S is equivalent to state N in the point of view of objects instantiated from *Patient*. So we mark state S as state N in class *Patient*. Of course the ancestor class has to be written to be able to handle the equivalent states. The technique we use is that we introduce a state transform map in the ancestor class and the descendant can register the equivalent states. After an object has called the function *getState* of a connected object and a mapped state exists for the result, the object uses the mapped state. We have to deal with how to describe it in the specification and how to implement it by generating concrete code.

We decided that we specify state mappings in UML class diagrams on specification level. We chose a form of expressing these mappings through stereotypes. We do it by using mapping functions. Mapping functions have a "mapping" stereotype and they have two parameters with type integer. These parameters have default values which describe the two states that have to be mapped. The code generator implements mapping functions so that they generate new mappings into the mapping vector of the ancestor class. In fact, these functions are "dummy" functions, because they only generate class level state mapping calls instead of generating real functions.

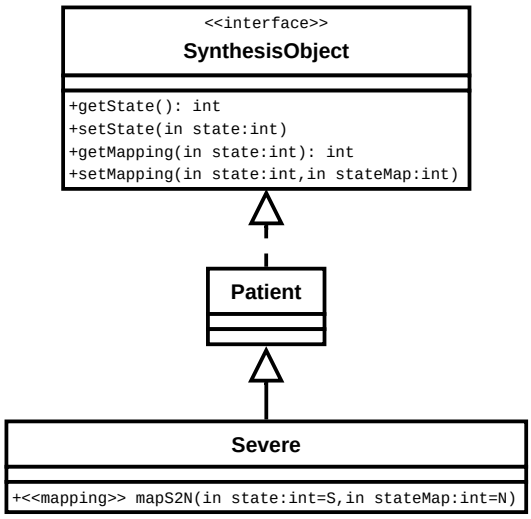

Figure 3.8: Synchronization part of the class diagram in the case of the modified example, where state S does not excludes state C.

If a state mapping is defined, we can replace mapped states in the synchronization code of the ancestor class, when examining the actual state of a connected object. This means that in our example synchronization code of class *Patient* does not need to be regenerated.

On implementation level the mapping vector has to be stored. We introduced two functions in *SynthesisObject*, which have to be implemented: *setMapping* and *getMapping*. Function *setMapping* gets two parameters for the mapping while *getMapping* receives a state parameter and returns with the mapped state or the input state if no mapped state exists. We consider the Java implementation of the solution. If a class implements this interface, it defines a private static variable M, which has a type *Map*, and implements the two mentioned function as we considered above.

According to these, a part of the class diagram of our example can be seen in figure 3.8. The generated code looks like:

```
public class Patient implements SynthesisObject {
  ...
  private static java.util.Map<Integer, Integer> M =
    new java.util.TreeMap<Integer, Integer>();
```

```
  public int getMapping(int state) {
    Integer ret = M.get(new Integer(state));
    return ret == null ? state : ret.intValue();
  }
  public void setMapping(int state, int stateMap) {
    M.put(new Integer(state), new Integer(stateMap));
  }
  ...
}

public class Severe extends Patient {
  ...
  public Severe() {
    // because of the mapping stereotype dummy function
    super.setMapping(S, N);
  }
  ...
}
```

In fact, it can happen that there are many classes in the system and the synchronization codes of all classes have to be modified because one of the classes is extended. In that case, state mapping can be used generally, so that it is valid for every class. That is, we can put the map M and the state mapping functions in class *SharedObject*. If this solution is used, then the descendant class has to call the function `SharedObject.setMapping` instead of function `super.setMapping`. In this case, it can happen that a new state has to be mapped to different states in different classes. So it has to be implemented so that state lists can be assigned to the new states, and if an objects uses the state mapping, it search for a proper state in the list.

3.5 Shared classes

In a concurrent system there are processes that use resources in parallel and they communicate with each other. Therefore, communication and using resources have to be synchronized. Shared classes are special entities of parallel object-oriented systems. In object-oriented paradigm, a concurrent

system can be regarded as a collection of autonomous active objects which synchronize and communicate through shared passive objects. If the used language supports object level parallelism, then shared objects have to be synchronized. That is, in the case of a shared class, an instance of the class is the shared resource, or it owns and handles the resources. Furthermore, if other entities use this instance, they can communicate through the methods and attributes of the object, so the instance of the shared class controls the synchronization, too.

Shared objects are instances of shared classes. If a language supports class level objects, in most cases a shared class can be considered as a shared object. Shared objects have methods that can run in several issues but these running methods reach the attributes of the same object. Therefore, reaching of methods of shared classes has to be synchronized. In fact, in the case of shared objects, methods can be consider as "virtual processes" and these virtual processes rival to reach the shared object as a resource or the resources provided by the object, and these processes communicate through the shared object. That is why we can say that shared objects have own synchronization and their classes have to be designed so that they can be used safely in a concurrent system, where methods of objects compete to reach the attributes. If a shared class is designed so that other objects can reach its methods safely in a concurrent environment, then the class is called "thread safe". We want to give a method that the synchronization code of shared classes can be designed and generated with.

Unfortunately, it can happen that a resource has to be reached in several steps, so sometimes the states of the processing have to be stored. So the methods of shared objects may be required to return with a partial result, that is, the state of the synchronization has to be stored. Similarly to the EJB objects in the J2EE (see [103]) we can distinguish stateless shared objects and objects with states. In this context, state means not synchronization state but computation state that determines the synchronization state. If an object is stateless, every method call initializes the synchronization in the initial synchronization state (see below).

3.5.1 Stateless shared classes

If a stateless shared class is used, then the methods of its objects do not store the synchronization state when they return. This means that the methods cannot continue the operations, since the object stores no states. Intuitively, in these cases if a method begins to work, it instantiates a proper synchronization class and sets its state to initial and places it to the synchronization (with the interconnection relation) and begins the computation work. If it needs a synchronization step, it calls the method *setState* of this synchronization class and if it is succeeded, it continues the computation, etc.

We can map a parallel system to the world of a shared class by introducing objects that use the instances of the shared class: these are the synchronization objects. The classes of the synchronization objects implement the *SynthesisObject* interface. These objects are assigned to the methods (so methods contain the computation code and synchronization objects represent the synchronization code). If the computation needs to be in a state that has to be synchronized, it calls the method *setState* of the proper synchronization object. For designing the synchronization of a shared class, we require that the synchronization class hierarchy has to be separated from the computation one, but the synchronization model is hidden, only the body of the shared class can use its methods. The synchronization model defines the synchronization of the methods of the shared class and we can specify it as a parallel object-oriented system (see below). The interconnection relation can be specified in the usual way by using the specification method that was introduced in Section 3.2.2, in which the methods of the classes of the synchronization structure can be used.

In fact, shared classes are usually designed to provide one kind of resources. So in most cases, the synchronization model contains only one synchronization class, which implements the *SynthesisObject* interface. Of course, there are cases, when the shared class provides a resurce that needs to be handled by more kinds of methods (e.g. if the shared class handles a database and it has two methods, read and write, then the shared class handles two types of synchronization objects). Besides, the synchronization

code of a shared class can usually be handled as a *one-class system* (e.g. because the synchronization code is synthesized after state amalgamation as one program, and we can handle it as a one-class system, if we do not split this program accordingly to the classes – because this is used only inside of the shared class).

If a service of a shared object (a public method) is invoked, it creates a new instance of the synchronization class, therefore, it is created in its initial state. After that, the computation can be started and if a synchronization step is needed, it can be implemented through the synchronization object. If an other method is invoked, the synchronization object has to be passed as an argument. If an other public method can be called from inside the class, and this method can be affected in the synchronization, then an other private function has to be created with a *SynthesisObject* type parameter and the body of the public function has to be placed in this new function. This new function can be called from inside the class. The body of the corresponding public function can be written so that it instantiates a new synchronization object and invokes the just introduced new function with *SynthesisObject* as parameter. For example consider the following public method.

```
public void f() {
    SynthesisObject o = new SynchronizationClass();
    <body of f>
}
```

If this function can be called from an other method, it can be transformed.

```
public void f() {
    SynthesisObject o = new SynchronizationClass();
    _f(o);
}
private void _f(SynthesisObject o) {
    <body of f>
}
```

In the case of the example above, only function `_f` can be invoked from inside the shared class.

Of course it can happen that before calling the implementation of the corresponding private function, function *setState* has to be invoked. It can be specified in the state diagram (see 3.6). At the end of a service, the synchronization object must be destroyed, so in languages with dynamic memory handling a deallocating statement is generated in the end of the function, while in the case of languages with garbage collector the function has to call method *destroySynchronization* of the synchronization object.

In the case of stateless shared classes we can use the technique introduced above, furthermore, we can often handle this case as a one-class system. The synchronization of a shared class must be specified as in the case of a parallel object-oriented system, and connecting the computation code with the synchronization code can be done in the same way as in the case of a shared system.

The main advantage of the stateless shared classes is that it cannot be seen in their export interface that they are implemented to be able to be used in a parallel environment.

3.5.2 Shared classes with states

There are cases when the full computation cannot be fulfilled in a service of a shared object, but it can fulfil only a part of an operation and after that it has to return with this partial result in order that an outer object can produce an other input based on this partial result. After that, the operation has to be able to be continued in the shared class. In some cases, these could be solved with callback functions, so if a service realizes that it needs some more data from the caller object, it calls a function, and this function produces the new input based on the partial result of the service and the service can continue its work. Unfortunately, there are cases, where it is not applicable, furthermore, forcing this structure upon developers gives a rigid structure of the program, which is not favourable.

Sometimes the state of the computation has to be stored, and it implies that the state of the synchronization has to be made persistent as well. For implementing this, a private vector – a synchronization vector – is intro-

duced in the shared class, which can store records with type integer and *SynthesisObject*. In Java it can be implemented as a Map with integer key type and *SynthesisObject* value type. Public methods that can return with a partial result, have an integer parameter. If they get zero in this parameter, they instantiate a new synchronization object and places it into the synchronization vector with a generated integer value, else it gets the stored synchronization object that belongs to the parameter value. Everywhere it needs to be synchronized, the method uses this synchronization object. If the method has to return with a partial result, it gives the integer identifier of its synchronization object as a return value. If the external object wants to continue computing the result of the task, it passes the integer identifier of the synchronization object of the service and execution continues from the synchronization state, which was passed. That is the reason that the key value have to be hard resolvable, so we usually chose a large random value.

Let us see, how to destroy the synchronization object. In the ideal case, the shared object knows if it executed the last step and it can call *destroySynchronization*. Unfortunately, there are cases when only an outer object knows that the work is done (e.g. if the outer object can accept a partial result). In these cases shared classes have to provide a method that destroys the synchronization objects.

3.5.3 Inheriting from a shared class

Assume that we make a shared class and we would like to use it by producing an other one with extended functionality. So we have to extend the existing one by using the tools of inheritance.

In these cases resources are provided by the shared object. If the descendant provides only new resources then it has to take care of only its own methods, which use the new resources. It can be controlled with an other synchronization system that is independent of the first one in the ancestor.

If the descendant wants to modify reaching the resources of the ascendant, it can happen that the synchronization has to be regenerated. There are two cases. In the first case, reaching the resources from the functions

of the descendant class introduces no new synchronization states. In this case, we can use the following technique: a new method is introduced in the ascendant class that instantiates the synchronization classes. This method has to be overridden in the descendant class so that it instantiates the new synchronization objects. Because the automaton of the new synchronization uses the same states in the same order, computation methods which call *setState*, do not need to be overridden. If the synchronization class can inherit from the synchronization class of the ascendant, then state mapping technique (see Section 3.4.4) can help to avoid using new synchronization objects in the computation methods of the ascendant. In the second case, new states are introduced of the synchronization of reaching the resources, so the computation methods of the ascendant have to be overridden because of the *setState* calls (this inheritance anomaly is difficult to solve).

3.6 Connecting the synchronization code to the computation code

Connecting the synchronization code to the computation code means generating synchronization calls to the computation code. In fact, synchronization states belong to computation activities. For example, in the case of the example of the surgery, state normal is the state where a human does not need to be synchronized with other objects, because if she is in this state, she is alone (at home by the assumption). In that reduced example the only place where more humans can be together is the waiting room or the examination room so these activities have to be synchronized. But these activities are connected to the computation code (for example, they use doctors as resources). So synchronization states describe situations that are relevant for a computation code fragment because other objects can play important role when working in these states. Therefore, these parts are protected by the synchronization code, that is, the connected synchronization object has to be in the proper synchronization state.

A natural way to handle synchronization states is through state diagrams

of the computation classes. State diagrams describe important states of classes and of course synchronization states are such important states. State changes have to be executed if an event is occurred and state changes can have conditions or actions. We have two tasks: computation states that describe also synchronization states have to be marked in order to know which states are synchronization states, and we have to label the edges with the relevant information, which is needed to generate the synchronization calls in the proper place of the computation code.

There can be state changes that do not modify the synchronization state, they describe only a new behaviour of the computation code. In such cases no synchronization is needed of course. So if a transition is executed, and its destination state is a synchronization state, which is not equal to the source synchronization state – the actual synchronization state –, then synchronization is needed. In our case this means that function *setState* has to be called. So a method is needed to specify how to call the synchronization code and an algorithm has to be given to implement the specification. It is very important in order to keep the consistency of the system.

In order to join computation states to synchronization states, we require the states of the state diagrams to be labelled with the corresponding synchronization states. This can be done by putting the synchronization state in square brackets after the name of the computation state, as it can be seen in Figure 3.9. The Synthesis algorithm checks the state diagrams and if a state diagram contains not all the states of its synchronization part, the algorithm gives a warning message. It cannot happen that more synchronization states belong to a computation state because we can require that in such cases computation states have to be split into more parts. But there can be computation states that no synchronization states belong to. The meaning of such states can be defined so that they do not change the synchronization behaviour (see below).

The next question is when to call function *setState*. Of course it has to be run when it is required to change the synchronization state. Function *setState* has a parameter, the destination state. The actual state is represented by the returning value of *getState*. So function *setState* has to be invoked if the

controlling is continued in a state that is connected to a synchronization state. In these cases the destination synchronization state is passed to *setState* as a parameter. The problem is that in many cases we do not know, what exactly a state change of the state diagram means. Transitions between states can be labelled with events that can have actions, which have to be executed, and conditions, which have to be checked before the state transitions (event [condition] / action). Unfortunately, in many cases these are only intuitive instructions to programmers that if an event happens, a state change has to be done, furthermore, in many cases the new state is not exactly represented in the state space. For example consider a seminar registration system. If a seminar is in a state "Open For Enrollment" and a student has enrolled for it but no more seats are available, it goes to state "Full". In this case the label of this state change can be: "student enrolled [no seat available] / addToWaitingList()". In this case the root cause is that a "student enrolled". But what does it mean at the level of the implementation? Code generators do not know from this intuitive information, when this state change has to eventuate and sometimes they do not know the full action set. In the point of view of the synchronization, the only action is calling *setState* (calling the synchronization code), so it does not need to be marked in the action part of the transition.

The problem is, how to describe the event part of the transition. Let us examine, what kind of events can trigger a change of the synchronization state.

1. After or before executing an own function.
2. After or before executing an own function, if after or before the execution the computation state of the object is connected to the synchronization state.
3. After changing a property that is relevant for the synchronization.
4. After or before executing a function of an associated object.
5. Other events that cannot be expressed (e.g. in the body of a method some commands require a state change before executing the first one).

In the first case, calling the function *setState* of the synchronization object of the entity is generated in the end or start position of the corresponding function. This can be denoted in the state diagram in the following way: transaction between states have to be labelled with functions, by using a special stereotype, namely "synchbefore" or "synchafter" and the name of the function. The parameter of the *setState* call is the destination synchronization state of the transition (in the state diagram). If more transactions are labelled with the same function, then the start states have to be taken into account by selecting the proper end state, that is the parameter of the generated call of *setState* depends on the returning value of *getState* (it is implemented with "if" statements).

```
<<synchbefore>>enrollStudent()
```

In the second case, a similar call of *setState* can be generated but it can have conditions, which we denote in the condition part of the transition with a special "synch" stereotype.

```
<<synchbefore>>enrollStudent()
  [<<synch>>getSeatsSize() >= maxSeats]
```

The evaluation of the conditions is not synchronized. If it is needed to be synchronized, then the system has to be designed so that the start state of the transaction belongs to a synchronization state that ensures the mutual exclusive condition evaluation.

In the third case, the condition can be written in the same way as in the previous case, but in this case no triggering event exists. In fact, there are several triggering events because every setter method of the relevant properties can generate a state transition. But from the variables and functions that are in the condition part, the names of the setter functions can be computed which the *setState* calls have to be generated in. So in these cases a *checkSetState* call is generated in the end of every setter function, attributes of which is in the condition part, where *checkSetState* checks if the conditions of the transition in the state diagram is fulfilled and if so, it calls the function *setState*. The signature of *checkSetState* is the same as *setState*. This can

be denoted by using a special notation: "«synch»attribute" followed by the condition part as in the previous case.

```
<<synch>>attribute [<<synch>>getSeatsSize() >= maxSeats]
```

It can cause problems if an attribute belongs to an other object, that is, the synchronization state of an object depends on the state of other objects. The best solution would be to register a listener that is called after the state of the other object has been changed. Unfortunately, setter functions of classes usually do not have listeners. In fact, it only causes problems if the class of the other object is produced by an other supplier. Because it can make the generation too difficult and this case can be transformed out, we do not consider such problems. So if there is a reference for an other object in the condition part of the synchronization transition, the generator algorithm shows a warning message that this issue has to be handled by hand. If the type of the other object is produced by the generator of the given system, then a callback function is generated into the end of the setter function and objects register their *checkSetState* functions to be called by the callback function when they connect to the appropriate other object.

In the fourth case, the generator has to know, where calling the method of the other object should placed. So the best that the designer can do is to enumerate the methods in which a call for the appropriate external function is possible. The synthesis algorithm generates a new function, namely "execute<originalName>" (the original name is the name of the function in the other object) and generates a call of this new method in the body of the enumerated functions. The body of the new method is a function call for the original method followed or preceded by a *setState* call. At specification level, it can be done by enumerating the corresponding functions between curly brackets, separated by commas. Of course these calls can have conditions.

```
<<synchbefore>>{forceStudentToEnroll}student.invited
  [<<synch>>getSeatsSize() >= maxSeats]
```

The example above means that in the *forceStudentToEnroll* function of the actual class a call for *executeInvited* is generated (after a condition checking

statement) and function body of *executeInvited* can be generated e.g. in Java like it follows.

```
void executeInvited(Student student) {
  setState(<destinationState>);
  student.invited();
}
```

The parameters of function *invited* are not filled in, because only the skeleton of the program is generated, parameters have to be given by the programmers. The "execute" functions always get the object, method of which has to be called, as a parameter. The types of these parameters can be searched in the declaration part of the calling function or in the class diagram.

The fifth case describes occurrences that cannot be handled in a common way. In these cases it may occur that a synchronization step is needed in the middle of a function, etc. In fact, these systems are not well designed, because the organization of the code is not correct, or it is about short code segments that are not worth handling in a difficult way. So these simple cases can be handled by the programmers or the design of the software can be modified so that synchronization calls are differentiated better from the synchronization code. Therefore we do not consider these cases.

There are cases when a synchronization step requires two or more state changes directly. For example if a process wants to use a resource, it has to require the object first (so it goes to the "require" state) and then it uses the resource (so it goes to the resource "using" state). Of course the process wants to use it immediately, but it has to require first. Therefore, the method that uses the resource, has to call function *setState* twice: the first one brings the object into the "require" state and the second into the "using" state. That means that there are functions the executions of which result two state changes.

We can map it in the following way: in some cases there is a transaction between two states so that the destination state (let it be D) is a synchronization state and the transaction contains no synchronization descriptions, but a previous transaction in the state diagram contains synchronization descriptions (let the start state of this transaction be S) and there is no other

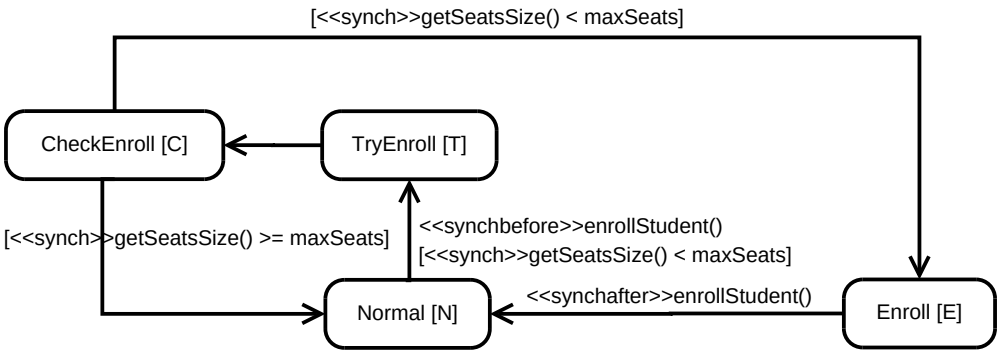

Figure 3.9: Connecting computation code to synchronization code by using modified state diagrams.

way from state S to state D. In such cases all the synchronization steps are generated to the appropriate method so it results the required source code (if only the condition parts exist on the transaction then they are used by generating the code). An example can be seen in Figure 3.9, where calling the function *enrollStudent* indicates more state changes before executing the body of the function. (If there is a state where two or more transitions with synchronization stereotypes start from, then conditions help to generate the proper calls of *setState*. Only transactions with synchronization stereotypes are taken into account to generate synchronization calls.)

The generated body of function *enrollStudent* in Figure 3.9 is:

```
boolean enrollStudent() {
  if ( getSeatsSize() < maxSeats ) {
    setState(T); setState(C);
    if ( getSeatsSize() < maxSeats ) {
      setState(E);
      // TODO: add your own part of code after this line
      // TODO: add your own part of code before this line
      setState(N);
    }
    else {
      setState(N);
      // TODO: add your own part of code after this line
    }
```

```
  }
  return false;
}
```

If a state can be reached from an other by using more function calls, then more "event [condition] / action" triplet can be used for a transaction. This is also valid for the synchronization notations. It cannot cause problems if the source state of a state change is not assigned to a synchronization state, because only the destination state is relevant in the code generation (except for the cases, if a state transition can be initiated from more then one states with the same function: in these cases the return value of function *getState* decides the proper call of *setState*). If the source state is assigned to a synchronization state but the destination state is not, and the proper state transition contains no synchronization descriptions, it means, that the object keeps its synchronization state, but it changes its computation state (if the state transition contains synchronization descriptions, then the proper end state has to be found to determine the parameter of the function *setState*).

The code generator has to take into account the dynamic synchronization behaviours. So if an object is in its initial synchronization state, then the algorithm checks the states that can be reached from the initial one for each transition and based on this, it determines the dynamic synchronization classes. If more synchronization classes are found, it searches for the transactions that lead to the different synchronization classes. If it cannot determine these transactions in the initial state, a warning message is printed, else it generates the proper instantiate statements of the synchronization classes.

If a program finishes its working (it steps into the final state) a call for *destroySynchronization* have to be generated.

The idea to connect the synchronization code to the computation code in some special cases can be found in [59].

Chapter 4

Extensions and improvements

In this chapter, we will show, how to extend our method to be more general and much more usable. We show examples that can be solved only by extending our algorithm and we introduce a specification method that can be embedded into a graphical designing tool, e.g. UML.

4.1 Extending MPCTL*

In this section, we will introduce an extension of the method, with which it will be possible to handle some special cases of object-oriented systems efficiently, too. A previous version of this extension can be seen in [57]. We will show our method through an example.

4.1.1 An example: Multiplier

Assume that we have N objects, and all of them can perform the "+" operation, and there are N objects, which can perform the "–" operation. It would be liked to perform the "$\cdot$" operation in the form "$a \cdot b$" where both a and b are natural numbers. There is a sender and a receiver object at the start and at the end of the system, respectively. The system works as it is described below.

- Organize the objects into a nonlinear pipeline system [101], [110].

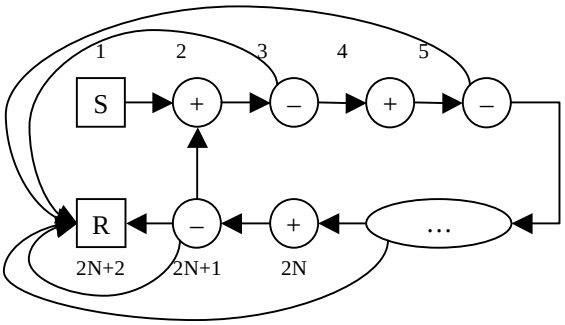

Figure 4.1: The functioning of the multiplier system.

  - Objects with even indices are "+"-objects (except for the last, because the last is the receiver).
  - Objects with odd indices are "–"-objects (except for the first, because the first is the sender).

- Every object gets three arguments: multiplicand, factor and partial solution.

  - The "+"-objects add the first argument to the third argument and send to the next "–"-object.
  - The "–"-objects subtract one from the second argument and send to the next "+"-object if the result is greater than zero or to the receiver if equal to zero.

- If the last "–"-object has a non-zero second argument, it sends its data to the first "+"-object.

The functioning of the system can be seen in Figure 4.1. As the figure shows, the receiver and the first adder object can receive data from more than one objects so they have to decide where to receive from. This problem can be handled in object-oriented environments but we do not have the formal tools. We developed an additional abstract semantic to handle this problem, so that it can be easily implemented.

4.1.2 The solution

In the following, we give the solution of the problem, and in the meantime we extend our earlier results.

In [57] we use an extension of spatial operators for solve this problem. You can read about this extension in [129], [130] and [131]. This extension is about introducing parameters for the spatial operators. These parameters filter the interconnections by using the parameters of spatial operators, which are predicate calculus expressions. Now, we use this extension in a special way. In the following we assume that the interconnection relation is antisymmetric. If we want to make the interconnection relation symmetric, we have to specify I so that $(o_1, o_2) \in I$ implies $(o_2, o_1) \in I$. By using this solution, specification of I is going more complex, so functioning of *makeI* is more expensive, but running of *setState* is faster. That is, why we chose this solution. We emphasize that specifying a system by using antisymmetric interconnection relation is a very hard task and it is easy to make mistakes.

In this approach, object o is related to an other iff o is equal to *leftObj* in the *SynthesisObjectPair*. By using this approach we have to modify the token capturing algorithm so that object a has to collect the token of (a, b) and (b, a) for every b in O (if the pair is in I), but examining of the state of an object b is needed only if $a\ I\ b$.

Let us partition the system into three pieces. We have two main kinds of processes: there are two objects at the ends (the sender and the receiver) and the remaining processes between the ends. In fact, it is a special kind of pipeline systems. We can solve the problem in two steps, as it can be seen in [129]: we synthesize the synchronization skeleton of the processes inside the pipeline and after that we generate the synchronization code of the sender and receiver objects, the first adder and the last subtractor. A solution is given for synthesizing a pipeline system in [129], so we only concentrate the "embedded system" and we do not consider the synchronization of the sender and receiver.

Figure 4.2 shows the class diagram of the system (the synchronization structure is not separated from the computation structure), which can help

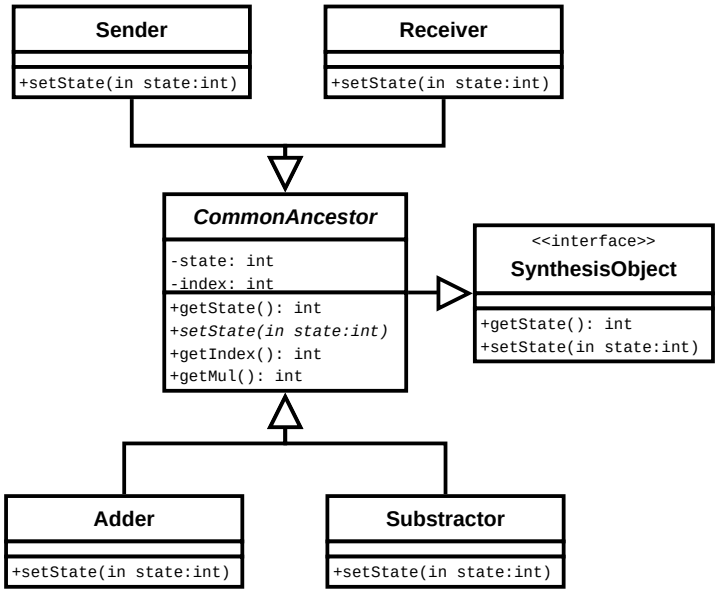

Figure 4.2: The class diagram of the system.

planning the states. Because we want to organize the objects, every object has a *getIndex* function. We assume that every object with an even index is an adder object and objects with odd indexes are subtractors. To ensure this condition is the task of the computation code. We could design our system so that it examines this condition but we leave it now. Objects have a function, named *getMul*, which returns with the value of the multiplicand (the second argument of the objects) in order to be able to examine whether it is equal to zero.

We define the states of the objects.

An adder object has the following states:

- N - Normal
- W - Work
- T - Try to read
- E - Try to send
- R - Read
- S - Send
- C - Check
- A - After send

A subtractor object has the following states:

- N - Normal
- T - Try to read
- R - Read
- C - Check
- W - Work
- E - Try to send
- S - Send
- A - After send
- D - Try to send the final result
- L - Send the final result
- F - After final result send

As it can be seen, we have to distinguish that a subtractor sends its result to the next adder or to the receiver. It's necessary because the relation I has to be changed dynamically depending on the state (see below).

Let us remark that every object has all of the 11 states but only the "–"-objects can be in the states D, L and F (see Section 3.3.2).

As we mentioned, the receiver and the first adder object can receive data from more than one other objects and all of the subtractors can send data to more than one objects. It has to be controlled to select an object by a subtractor to send data to and to select an object by the receiver or the first adder to receive data from.

The antisymmetric interconnection relation and parametric spatial operators are used for making an order for the data receiving in the receiver and in the first "+"-object in the following way: if an object is going to state D then it places itself and the receiver into the relation I. The receiver is connected with only the first object that is in one of the states D, L or F. This is controlled by the antisymmetric interconnection relation. Only the connected objects are synchronized, so subtractors in states D, L and F have to be connected to the recevier and not to the next adder. The case, when the last subtractor wants to send the data to the first adder, has to be handled similarly.

4.1.3 Extension of the spatial operators

A process cannot receive data from more than one processes simultaneously. The sending priorities are handled through the interconnection relation I and parametric spatial operators. But we have the problem coming from the fact that an object can receive data from more that one objects. If the

object is receiving data from an other object but a third object wants to send data too, then this object has to be blocked. It can be achieved via the interconnection relation. If interconnection relation were symmetric, it could be solved so that if an object wants to send data to the receiver rather than its direct following, it removes the pair of itself and its direct following from I and places a new pair of itself and the receiver into the interconnection relation. But if so, then the next object has to be blocked because nobody is there, who can send data to it. However, it will not be blocked, because of the semantic of the operator $\otimes$.

Because we have made I antisymmetric, every object controls its own relation. So if an object a is in the interconnection relation with its following object (let it be b), then it can happen that object b has placed a pair (b, a) in I because it wants to receive data from a. If a removes the pair (a, b), an other pair, (b, a) still is in I, so b is connected to a. But in this case b has to be blocked, too, because a does not send anything to b, but b does not know it. So b has to be blocked if b is connected to a but a is not connected to b. Therefore, we introduce an operator that provides the following: if an object a is connected to b but b is not connected to a then a must be blocked, even if the transition is enabled by the other constraints.

A "boolean" parameter of the spatial operators is introduced (at specification level) which controls the result, depending on the connection set of the connected objects: if the parameter is false, the conditions cannot be evaluated true if there are connections that have no inverse pairs in I (for details see [57]).

On implementation level, in every transition two kinds of the $\otimes$ operator can be used (of course even both can be used together connected by logical operators):

- weak $\otimes$ operator evaluates true if the inverse pair of the relation cannot be found in I, so the transition is enabled – this corresponds to the *true* value of a spatial operator's parameter at specification level,
- strong $\otimes$ operator (marked with $\otimes\otimes$) evaluates false if the inverse pair of the relation cannot be found in I, so the transition is not enabled

– corresponds to the *false* value of a spatial operator's parameter at specification level.

Let us see the extended MPCTL* specification of the synchronization of the embedded system (we assume that senders can be only in states W, E, S, A and N, while receivers in state N, T, R, C and W, and so we do not need to take care of them).

1. Initial state (every process is initially in its normal state):

$$\bigwedge_i N_i$$

2. It is always the case that any move P_i makes from its R state is into its C state, and such a move is always possible (and similarly for the states W, S and L):

$$\bigwedge_i AG(R_i \Rightarrow (AY_i C_i \wedge EX_i C_i)) \qquad \bigwedge_i AG(W_i \Rightarrow (EX_i(E_i \vee D_i)))$$

$$\bigwedge_i AG(W_i \Rightarrow (AY_i(E_i \vee D_i))) \qquad \bigwedge_i AG(S_i \Rightarrow (AY_i A_i \wedge EX_i A_i))$$

$$\bigwedge_i AG(L_i \Rightarrow (AY_i F_i \wedge EX_i F_i))$$

3. It is always the case that any move P_i makes from its N state is into its T state – but such a move is not definitely possible (and similarly for the states T, C, E, A, D and F):

$$\bigwedge_i AG(N_i \Rightarrow AY_i T_i) \qquad \bigwedge_i AG(E_i \Rightarrow AY_i S_i)$$

$$\bigwedge_i AG(T_i \Rightarrow AY_i R_i) \qquad \bigwedge_i AG(A_i \Rightarrow AY_i N_i)$$

$$\bigwedge_i AG(C_i \Rightarrow AY_i W_i) \qquad \bigwedge_i AG(D_i \Rightarrow AY_i L_i)$$

$$\bigwedge_i AG(F_i \Rightarrow AY_i N_i)$$

4. P_i is always in exactly one state of the state set:

$$\bigwedge_i AG(N_i \equiv \neg(T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$

$$\bigwedge_i AG(T_i \equiv \neg(N_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$

$$\bigwedge_i AG(R_i \equiv \neg(N_i \vee T_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$

$$\bigwedge_i AG(C_i \equiv \neg(N_i \vee T_i \vee R_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(W_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(E_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee S_i \vee A_i \vee D_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(S_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee A_i \vee D_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(A_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee D_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(D_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee L_i \vee F_i))$$
$$\bigwedge_i AG(L_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee F_i))$$
$$\bigwedge_i AG(F_i \equiv \neg(N_i \vee T_i \vee R_i \vee C_i \vee W_i \vee E_i \vee S_i \vee A_i \vee D_i \vee L_i))$$

5. Liveness: if P_i is in state N, then some time it will reach state T (and similarly for the states T, C, E, A, D, and F):

$$\bigwedge_i AG(N_i \Rightarrow AFT_i) \qquad \bigwedge_i AG(E_i \Rightarrow AFS_i)$$
$$\bigwedge_i AG(A_i \Rightarrow AFN_i)$$
$$\bigwedge_i AG(T_i \Rightarrow AFR_i) \qquad \bigwedge_i AG(D_i \Rightarrow AFL_i)$$
$$\bigwedge_i AG(C_i \Rightarrow AFW_i) \qquad \bigwedge_i AG(F_i \Rightarrow AFN_i)$$

6. A transition by a process cannot cause a transition by another one:

$$\bigwedge_{ij} AG((E_i \Rightarrow AY_jE_i) \wedge (E_j \Rightarrow AY_iE_j))$$
$$\bigwedge_{ij} AG((N_i \Rightarrow AY_jN_i) \wedge (N_j \Rightarrow AY_iN_j)) \qquad \bigwedge_{ij} AG((S_i \Rightarrow AY_jS_i) \wedge (S_j \Rightarrow AY_iS_j))$$
$$\bigwedge_{ij} AG((T_i \Rightarrow AY_jT_i) \wedge (T_j \Rightarrow AY_iT_j)) \qquad \bigwedge_{ij} AG((A_i \Rightarrow AY_jA_i) \wedge (A_j \Rightarrow AY_iA_j))$$
$$\bigwedge_{ij} AG((R_i \Rightarrow AY_jR_i) \wedge (R_j \Rightarrow AY_iR_j)) \qquad \bigwedge_{ij} AG((D_i \Rightarrow AY_jD_i) \wedge (D_j \Rightarrow AY_iD_j))$$
$$\bigwedge_{ij} AG((C_i \Rightarrow AY_jC_i) \wedge (C_j \Rightarrow AY_iC_j)) \qquad \bigwedge_{ij} AG((L_i \Rightarrow AY_jL_i) \wedge (L_j \Rightarrow AY_iL_j))$$
$$\bigwedge_{ij} AG((W_i \Rightarrow AY_jW_i) \wedge (W_j \Rightarrow AY_iW_j)) \qquad \bigwedge_{ij} AG((F_i \Rightarrow AY_jF_i) \wedge (F_j \Rightarrow AY_iF_j))$$

7. Data flow control: a process in state T waits for the previous process to reach state A and a process in state C waits for the previous process to leave A (and similarly the two other rules):

$$\bigwedge_{ij}(false)AG((E_i \wedge C_j) \Rightarrow \neg EX_i true$$

$$\bigwedge_{ij}(true)AG((N_i \wedge W_j) \Rightarrow \neg EX_i true \qquad \bigwedge_{ij}(false)AG((A_i \wedge \neg C_j) \Rightarrow \neg EX_i true$$

$$\bigwedge_{ij}(false)AG((T_i \wedge \neg A_j) \Rightarrow \neg EX_i true \qquad \bigwedge_{ij}(false)AG((D_i \wedge C_j) \Rightarrow \neg EX_i true$$

$$\bigwedge_{ij}(false)AG((C_i \wedge A_j) \Rightarrow \neg EX_i true \qquad \bigwedge_{ij}(false)AG((F_i \wedge \neg C_j) \Rightarrow \neg EX_i true$$

8. A process cannot be in state W if a connected one is in state R or C (because of the dynamic changes of I: a process is not allowed to begin sending data to an other process while the other is receiving data from a third one – it is checked by function *checkConsistency*):

$$\bigwedge_{ij} AG(\neg(W_i \wedge (R_j \vee C_j)))$$

9. Always there is a possible step:

$$AGEX\,true$$

Specification of I is quite difficult. Because all object in the system is instance of a class that is a descendant of class *CommonAncestor*, we can use $\uparrow$ for class *CommonAncestor* instead of *SynthesisObject*.

$(\uparrow,\uparrow)\bigwedge_{ij}(N_i \vee T_i \vee R_i \vee C_i) \wedge ((i.getIndex() = 2 \wedge j.getIndex() = 2N+1 \wedge j.getMul() > 0) \vee ((i.getIndex() \neq 2 \vee (\neg\exists k)(k.getIndex() = 2N+1 \wedge k.getMul() > 0)) \wedge i.getIndex() = j.getIndex() + 1))$

$(\uparrow,\uparrow)\bigwedge_{ij} W_i \wedge ((i.getMul() > 0 \wedge i.getIndex() = 2N+1 \wedge j.getIndex() = 2) \vee (i.getMul() = 0 \wedge j.getIndex() = 2N+2) \vee (i.getIndex() \neq 2N+1 \wedge i.getMul() > 0 \wedge i.getIndex() + 1 = j.getIndex()))$

$(\uparrow,\uparrow)\bigwedge_{ij}(E_i \vee S_i \vee A_i) \wedge i.getMul() > 0 \wedge ((j.getIndex() = 2 \wedge i.getIndex() = 2N+1) \vee (i.getIndex() \neq 2N+1 \wedge i.getIndex() + 1 = j.getIndex()))$

$(\uparrow,\uparrow)\bigwedge_{ij}(E_i \vee S_i \vee A_i) \wedge i.getMul() = 0 \wedge j.getIndex() = 2N+2$

$(\uparrow,\uparrow)\bigwedge_{ij}(D_i \vee L_i \vee F_i) \wedge j.getIndex() = 2N+2$

Notation X_i means: `i.getState() = X`. (The formulae above are connected with the "or" operator.)

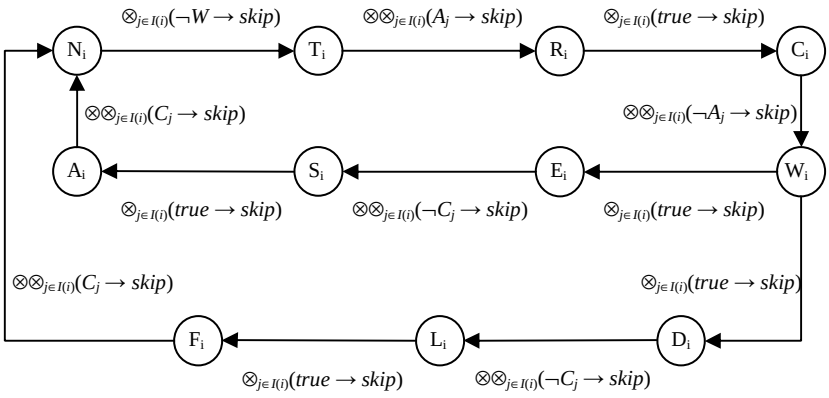

Figure 4.3: The synthesized synchronization skeleton of the system.

The resulted skeleton of the synchronization code, after running the algorithm of Attie and Emerson (in [7]), can be seen in Figure 4.3.

Operators are strong before receiving data so if nobody sends data to an object but it wants to receive, it has to be blocked (and similarly for sending data). Of course *setState* is generated so that if it examines the conditions that are related to an other object and the conditions are true but the other object is not connected to the actual, it sets the variable *succeed* to *false* and no state transition is possible.

Because I depends on function *getState*, *getMul* and *getIndex*, the corresponding setters, *setState*, *setMul* and *setIndex* are generated so that they call *makeI* after fulfilling these operations. If an object steps into state N, it connects itself to the prior, if an object steps into state W, it connects to the next object. If an object finished the multiplication, it connects itself to the receiver instead of the next object, so the next object cannot receive data because of the strong operators. Furthermore, if the last subtractor has a non-zero multiplicant, it connects itself to the first adder instead of the receiver, and because of the specification of I, the connection of the sender and the first adder is removed, and the connection of the first adder and the last substractor is added. After the last subtractor steps in state N, that is, the communication is finished, it removes its connection to the first adder, which causes that the original connections of I are restored.

Because of the specification condition 8, a *checkConsistency* function has to be generated and *makeI* ensures that I cannot be changed if this function returns false. For example, if an object wants to connect itself to the receiver (it can be done in state W because objects decrement the value of the multiplicand in state W), this function does not allow this object to connect itself to the receiver if the receiver receives data from an other object, that is, if it is in one of the states R or C.

4.2 A graphical tool for specifying the synchronization

We have extend the Unified Modeling Language with a new diagram class. A *synchronization diagram* can be created for every system. Restrictions can be formulated for the states of the system in the *synchronization diagram*. Because it is the extension of UML, all of the states can be used that are specified in the class diagram at the appropriate classes. Unfortunately, all the states can be used and no checking can be made because of the state union (any objects can be connected to others with different types and different state sets and they can effect each other). An earlier version of synchronization diagrams can be found in [59].

We designed synchronization diagrams so that the important features that can be formulated in MPCTL* can be expressed by using synchronization diagrams. So, primitive building elements are the bases of this type of diagrams. These base elements ensure that the usual constraints of MPCTL* can be formulated. You can read about synchronization diagrams and its base building elements in [59] in details. Besides, for the sake of the better usability we introduced several abstract building elements. For example, we always have to include a logical expression that describes that a state excludes the others – it has to be done for every state. Describing this with primitive elements makes the diagram very large and hard to understand. So we introduced a simple element that this constraint can be described with. Of course, such kind of an element is compiled to several temporal logic for-

mulae (according to the number of states). An other example is that most systems require an object not to change the state of the others. Instead of several building blocks, we formulate this in a simple set of boxes. Figure 4.4 shows the synchronization diagram of a "read/write" system. In a read/write system there are readers and writers, which use a database. If a writer writes the database, no other writers and readers can use it. If a reader reads the database, other readers can use it, but writers are disabled. In Figure 4.4 the box with label "[ME]" represents a mutual exclusion specification for states Normal, Try and Write. This box is compiled to several MPCTL* formulae. The box with label "[SingleState]" expresses that objects can be in only one state of the enumerated states and box with label "[NoChangesOther]" specifies that an object cannot change the states of other objects. The dotted style bounded oval box expresses that all objects are created in state Normal. The other symbols formulate that readers eventually can step into state Read from state Try and from state Read always exists a step to state Normal, while states Write and Read exclude each other. Because of the specification there are two possible transitions from state Try, so we generate the MPCTL* formulae so that they describe this. It is important, because these features can be formulated hard by using temporal logic formulae. You can read about the main building blocks of synchronization diagrams in Appendix B.

In order to be able to draw synchronization specifications, an extension of *Dia* [29] was developed. Diagrams that are written in *Dia* are saved in XML [33], [146] files. These XML files can be parsed and transformed to an other XML file that contains MPCTL* formulae that are coded in a special XML form. The code generator algorithm receives input of this kind, accordingly an XML file that contains MPCTL* formulae.

The input of the tool is a *Dia* file, or its decompressed content, which is XML data. The file must contain a graphical specification that is built from the elements which were described in the previous sections. The output is an XML description for each of the given specification elements.

The handling of the interconnection relation is left unchanged. Interconnection relations are usually simple but type dependent. So we can often

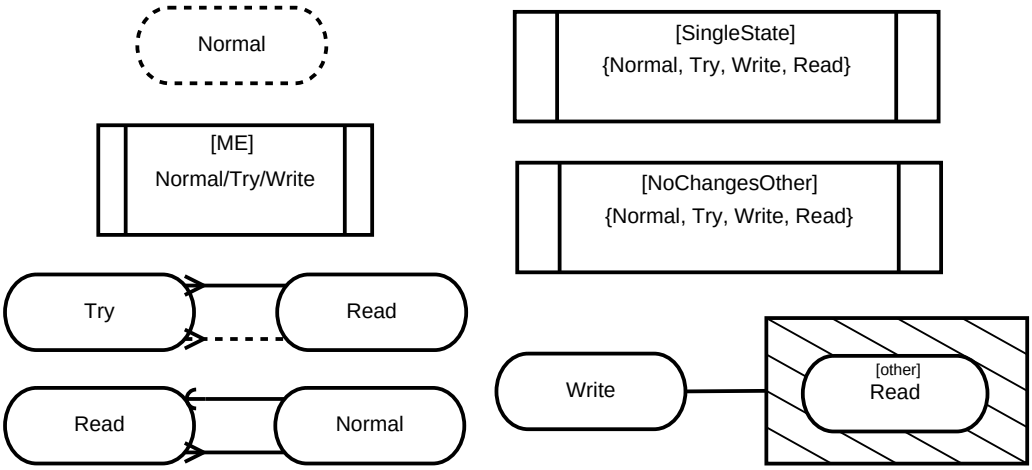

Figure 4.4: Synchronization specification of a read/write system.

formulate the requirements of the interconnection relation for every type with some simple predicates. Therefore, we chose to specify interconnection relation in a special comment tag of classes (in the class diagram). CASE tools that support making UML diagrams usually provide a method to connect special tags to classes in the class diagram. For example in *Dia* there is the possibility to attach *Comment*s to classes – most CASE tools provide this possibility. Since we extended *Dia*, we specified placing the interconnection relation for *Dia*. We introduce a dummy class, which is marked with a stereotype "«interconnection»", and its comment field contains the interconnection relation of the system (and in the comment field stereotype «interconnection» has to be written, too). For example specification of the interconnection relation in the case when objects are connected iff they are neighbours (and it is true for instances of class *IndexedObject*, which provides method *getIndex*) is the following.

«interconnection»

(IndexedObject, IndexedObject)$\bigwedge_{ij}$ i.getIndex() = j.getIndex() + 1 $\vee$ i.getIndex() + 1 = j.getIndex()

4.2.1 Introducing class groups

In the case of many systems there are parts, objects of which collaborate stronger, while collaboration with other objects is rarer. In these cases it is not worth handling the whole system as a coherent one, because it is better not to handle all the states that have no business with each other. Following [59] we can partition the system to *class groups*. (Process groups are introduced in [129], which is a more general concept, but we consider a usable concept for classes.) Objects in class groups collaborate strong inside the group, while the outgoing communication is pure. Synchronization is generated for every class group separately. (We assume that the state sets of the different groups are disjunct.) After generating synchronizations of all the groups, communication of groups can be specified. Groups usually communicate through special classes, which are designed for transacting the communication between groups. So the specification of synchronization can be defined through these classes (of course all classes of groups can be responsible for the communication with other groups but usually only a few classes do that). After generating the synchronization code between groups, there are objects that have more synchronization codes (a synchronization for communicating within the group and a code to communicate with outer objects). So there are classes, synchronizations of which have to be produced from more automatons. We merge these automatons: if there is a state transition which has different conditions in the different cases, we union these conditions via the $\oplus$ operator: joining $A_i \xrightarrow{\otimes_{j\in I(i)} f} B_i$ and $A_i \xrightarrow{\otimes_{j\in I(i)} g} B_i$ results $A_i \xrightarrow{\otimes_{j\in I(i)} (f\oplus g)} B_i$. It can be done because there are different states in the different class groups, so operator $\oplus$ can be applied. Of course new states are not created during joining automatons, states of the other group can appear in only the conditions of transitions.

Class groups are handled in the graphical representation so that every group has an own synchronization diagram, furthermore, classes that synchronize between groups get further diagrams. Enabled states are defined for classes. So the generation of the synchronization skeleton for a class is through keeping only the states that are enabled for the class.

In the example of Section 4.1.1 there are three groups: {Sender}, the embedded system and {Receiver} (the first and last groups contain only one element, so there is no synchronization inside the groups).

4.3 Effectivenes of the generated code

We have shown a method that results in a generated synchronization code of some object-oriented systems. If a part of a program code is automatically generated, a question arises immediately: how good is the code? Of course goodness has several components. One of these is the runnig speed, an other is de complexity of the code, which is a measure of how difficult is to maintain the program. You can read about efficiency of programs in [40], [50], [75], [77], [100], [114] and [145]. In the following, we consider the runnig speed (but improving the runnig speed sometimes makes the code simpler so it can affect other aspects). Of course the code that is generated by our algorithm cannot be optimal. If a general algorithm is given to synthesize a program code, all possible specialities never can be taken into account.

Synchronization has two levels. The first level is the synchronization itself, when connected objects have to be synchronized. The second level is to keep the validity of the interconnection relation, where the generated code has to be decide, which objects are connected – this is also an expensive task.

Possibilities of solving the problems of the first level can be found in [55] and [129]. We only consider controlling the optimization at the level of diagrams and optimizing of handling I.

4.3.1 Stereotypes of synchronization diagrams

In [129] there is a solution of the optimization task based on MPCTL* patterns. Although optimization with patterns can result in a good improvement of performance, it is hard to use (users have to know MPCTL*). So we introduced an optimization type in a more abstract layer, in the synchronization diagram. This solution uses the solution of B. Ugron [129] but it makes easier to use optimization rules and creating new optimization patterns. The

described solution is a possibility of introducing optimization but we have not implemented it. We considered the optimization by using stereotypes in [55].

The new method of optimizing synchronization code is using stereotypes. A stereotype is a diagram which consists of elements that can be used in synchronization diagrams. Stereotypes are stored in a database in the precompiled MPCTL* format (this is the MPCTL* pattern [129]) but the diagram is also stored in XML [33], [146] format. This means that if we compile the stereotype (which is a small synchronization diagram) then we get the corresponding MPCTL* code that is stored in the same record.

Stereotypes have optimized codes. If we make a database that stores stereotypes with optimized codes and stereotypes have the corresponding MPCTL* formulae then the generator that makes the MPCTL* specification for the synchronization diagram compiles the sytereotype to the corresponding MPCTL* formulae (which is an MPCTL* pattern [129]). So after code generation, an optimization algorithm that works with MPCTL* patterns, and was developed by B. Ugron [129], can be used. The optimizer finds the pattern and it decides whether it can be used. If the pattern can be used, then the optimizer returns true and we can use our own rules to change the corresponding part of the synchronization code to the optimized code. Because the optimization can depend on the interconnection relation, the specification of the interconnection relation has to be stored with the stereotype. It is very important that the optimizer returns true only if the interconnection relation is the same as in the pattern. Consider the case of the mutual exclusion as an example. If every object is connected to each other, the problem of the mutual exclusion can be solved with only one semaphore on implementation level. If objects are connected only to their neighbours, then semaphores should be used for every object pair. In this case it can be a problem that objects can be created and destroyed dynamically. If an object is created and connects to one or two other objects, the corresponding semaphores have to be created and initialized. Optimized codes can be implemented in many languages that can be handled by the code generator.

Now we describe our own rules to change the proper parts of the synchro-

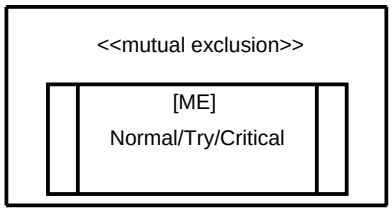

Figure 4.5: Stereotype for the mutual exclusion. It can be used only with the corresponding interconnection relation, that is shown in the comment part of the pattern. For example, it can be $(\uparrow, \uparrow) \bigwedge_{ij}$ true.

nization code to the optimized code. There are optimized programs stored with the pattern. More programs can belong to a pattern, but every program has to have an own start state and end state descriptor. These mean that if a pattern is applied then the optimized code will be run if a state transition has to be made from the start state to the end state. So if the optimizer returns true, we generate new functions with the name by concatenating the start state and the end state, and the body of these new functions are the optimized codes that belong to the states. After that, for all end states that are defined in the pattern, calls of *setState* are replaced with an optimized function so that the actual state is decided with the *getState* function and the corresponding optimized function can be chosen based on the actual and end states.

We remark that the optimizer decides to use the pattern only if the related states are not affected by other states (it is not a trivial task but we do not consider this here). For example, if the stereotype of the mutual exclusion is used but the designer intorduces readers with state *Read* and this new state excludes the *Critical* state then the stereotype cannot be used.

Patterns can be used so that developers load them when designing the synchronization by using synchronization diagrams. Figure 4.5 shows the stereotype of the mutual exclusion, but we emphasize that it can be used only with the corresponding interconnection relation.

Creating stereotypes

Users can create their own stereotypes. Users can mark some parts of the synchronization diagrams as stereotypes that describe special cases of synchronizations. After that, users can save these stereotypes. During the save algorithm stereotypes are compiled into MPCTL* formulae and are saved into the database in a transformed form (e.g. concrete states will be replaced to variables, etc.). Of course, stereotypes are saved with the actually used interconnection relation.

After a user saved the new stereotype, she has to define the optimized program code. It can be done so that she defines the functioning of the system for every state pair by using activity diagrams and compiles the program code to several languages. It cannot be done in all cases, in that cases she has to write the code by hand in many languages. In some cases, it can be a very large and hard task so our goal is to find as many stereotypes as it is possible so that users can use built-in stereotypes.

4.3.2 Optimized reaching of the interconnection relation

There are lots of cases where the generated algorithm examines the interconnection relation in order to decide, whether object pairs can be removed or placed. As we mentioned before (in Section 3.2.3), if the interconnection relation has to be changed, the algorithm rechecks all the existing connections and searchs for new ones only if quantifiers are used in the specification of I. So without quantifiers, handling relation I may be more faster.

Besides, there are cases when calling *makeI* can be skipped. Considering the example in Section 4.1.1, there are four states (N, T, R and C), where objects are connected to the previous ones and in other states (W, E, S, A) objects connect to the following ones. Since the changing of connections happens after changing the actual state, invoking of *makeI* is placed in function *setState*. It can be seen that running of *makeI* results changes in I only after every fourth execution but every execution is very expensive. So a more efficient code can be produced if calling *makeI* happens only if the new state

is N or W.

The solution is to give the possibility to define exceptions of maintenance of the interconnection relation. So designers can forbid the execution of function *makeI* and the generated program is going to be more efficient. The place, where exceptions can be defined, is the class diagram with a special "«iexception»" stereotype in the comment blocks of classes. For example in the case of the multiplier system, the following can be written in the comment block of class *CommonAncestor*.

```
<<iexception>>getState() != N && getState() != W
```

This way of formulating exceptions is not a language dependent syntax, but it is our own syntax. Of course, if the code is compiled e.g. to Delphi, it is compiled to the following code.

```
procedure CommonAncestor.setState(Value: Integer);
begin
  ... // generated body of procedure setState
  if (getState = N) or (getState = W) then makeI(Self);
end;
```

By using the generated code above, function *makeI* runs rarer, so the software computes its result faster.

4.3.3 Optimized execution of *setState*

There are cases when *setState* has to be executed twice and no other actions are between the two execution. For example in Section 3.6 the generated code of function *enrollStudent* calls `setState(T)` followed by `setState(C)`. Every state transition needs a token capturing which is an expensive task, and in many cases both state transitions can be executed. So we can optimize the execution of *setState* in the following way. Function *setState* gets an array of states as a parameter. It collects the tokens and it tries to execute the first state change. If it is succeeded then it tries to set the second state, and so on. If one of the states cannot be set, it releases the tokens and is blocked on a semaphore. In this case we can call the function *setState* in the following

way: `setState({T, C})` (it cannot be written in Java, we use it because of its shortness). It can be applied in many cases and it is a useful optimization because token capturing is very expensive.

Chapter 5

Related Works

Describing the expectations of the synchronization of a system can be done by invariants. Synchronization can be drafted in a very rough way so that processes have to be blocked if the conditions of their work are not satisfied. By using this approach only simple program codes belong to the synchronization: codes that block the execution of a process, evaluating a condition or waking a blocked process up. Of course the synchronization can be more complex, but it shows why synchronization code can be synthesized by using invariants and first order predicate calculus.

G. R. Andrews has given a method that is able to synthesize the synchronization of a system from synchronization invariants that are described in first order predicate calculus [4], [73]. This method works with global invariants but uses local invariants as well that are valid only in a specified program state. Developers have to specify the atomic statements, that is, statements that have to be executed without blocking. Based on these, a correct abstract program can be synthesized by using weakest precondition calculus. For implementing the abstract program, a general method is introduced, named "passing the baton". This method uses semaphores for implementing the statements.

This kind of synthesis is not so general because liveness conditions cannot be formulated in the specification, and the synchronization code is not separated from the computation code so they can be handled harder. Besides, it is a good method and generates an efficient synchronization code, so variants

of this method are still in use. M. Dwyer et al. have worked on developing synchronization code of classes, based on the global invariant (GI) approach [34]. They develop the component code by the Unified Process in UML and the synchronization code is developed as aspect code, which is later woven into the component code to produce complete object-oriented programs in object-oriented languages. They use different techniques for the synchronization and communication depending on the language (semaphores in C++ by using POSIX threads, Java synchronized blocks, etc.).

There are several methods based on temporal logic specifications. These algorithms usually use a tableau method for synthesizing the code. Synthesis methods can be found in [88], [89] that are based on a linear time temporal logic specification. There is a method for synthesizing the synchronization of recursive programs in [87].

The earlier synthesis methods suffer from the state explosion problem, so the solutions were developed that can avoid this by introduction some constraints. A solution can be found in [39], which uses only similar processes. This solution is able to generate the synchronization of systems, where every process is connected to each other. Ebnenasir and Kulkarni extended the method of Attie and Emseron [7] so that other liveness properties can be added to the specification without regenerating the full synchronization (only the new parts are generated and joined into the old skeleton) [36]. There is an extension of [7] in [76] that can be used for automated synthesis of multitolerant programs, i.e., programs that tolerate multiple classes of faults and provide a different level of fault-tolerance to each class, and there is an another approach to synthesize fault-tolerant programs in [5].

If a synchronization skeleton is generated by using the method of Attie and Emerson [7], then an abstract automaton has to be implemented. This implementation has to evaluate the conditions of the state transitions, which can be very expensive. Attie and Emerson introduced a method, which transforms the synchronization skeleton to an atomic read/write model [6], where only atomic variable reads and writes are used. Unfortunately, this method needs to make the full global state transition diagram and that is very expensive (in fact, exponential).

There are other methods that are not based on a tableau method. Lamsweerde and Sintzoff use fixpoint equations in [81] to synthesize the synchronization of programs. In [85] synthesis is solved by using algebraic tools.

Algebraic specifications are very useful to specify abstract classes (and there are methodologies to implement the functions, and methods exist to verify the implementation). There are methods to specify the synchronization of a shared abstract class with precedence relations [73]. This method distinguishes three phases of executing a method: request (req), enter (ent) and exit (ex). The specification language offers possibilities to define the precedence of these phases of (different) methods. There is a method to implement the synchronization code given by precedence relations [73]. The method introduces counters and derives gate conditions over the counters. These conditions are checked by await statements (see above), which can be implemented by using the passing the baton technique.

An other specification method for synchronization is using Petri Nets [106] or Object-oriented Petri Nets (OOPNs) [43]. OOPNs are dialects of high-level Petri Nets that join together object orientation and Petri Nets. An OOPN class consists of an object net, a set of method nets, and a set of synchronous ports. Object-oriented Petri Nets are useful to describe the parallel behaviour of object-oriented systems. Petri Nets are usually used for checking some properties of the synchronization code. This can be done so that Petri Nets can be generated from the program code and the generated Petri Nets are analysed. Besides, Petri Nets can be used to specify the synchronization, and based on this, the synchronization code can be generated. Mortensen has shown an automatic code generation technique for generating program code from general Coloured Petri Nets [99]. He has written the principles to generate the code but has not given the concrete algorithm. Furthermore, this algorithm is a very general algorithm and does not concentrate on the synchronization code.

As we mentioned before, most software engineers find standard temporal logics difficult to understand and use. Dillon et al. have given a solution for designing the synchronization specification of parallel software with a graphical tool by using graphical interval logics (GIL) [31], [79]. The authors

made a Graphical Interval Logic Toolset which is able to check whether a formula set can be deduced from the specification (automatic proof – so users can check some important behaviour, e.g. liveness properties, etc.) and the tool provides model generator features. Model generator creates an automaton. Unfortunately, the generated automaton describes the full system instead of the automaton of the individual processes.

An other way to design the parallel behaviour of an object-oriented system is by using UML diagrams [15], [90], [108], [135]. There are diagrams in UML2 that help to plan parallel systems. CASE tools [61], [104], [108] can generate some parts of the program code by using some of the diagrams. Users can use sequence diagrams to design the software to run some computations in parallel. This type of diagrams can be used for describing that two or more threads have to be finished to be able to continue the computations. So code generators can generate "fork" and "join" like statements. Besides the usual way to specify the parallel behaviour of a system in UML2 is through using activity diagrams. Activity diagrams can be used similarly as sequence diagrams expect for that explicit fork and join nodes can be used. So it can be defined that the computation has to be continued in more threads and there can be designed synchronization states where threads are joined. Unfortunately, synchronization regions, mutual exclusion and conditional synchronization are not supported in the current UML2 semantic model ([68]).

In [68] you can read about an extension, in which UML-based approach is proposed to specify secured, fine-grained concurrent access to some shared resources in a secure way. The authors extended the UML activity diagrams to support concurrency features and use normal UML protocol statemachines to define the access protocol for shared resources. Unfortunately, no implementation algorithm has been made, and for large systems the concurrent activity diagrams are too complex and hard to understand. Furthermore, this method concentrates only for designing the local synchronization of shared objects.

You can find an aspect-oriented [80] methodology in [98] for specifying the synchronization of systems in UML. This approach uses global invariants

(GI). Finding a global invariant is not an easy task, so GI patterns are introduced to compose complex GIs but GI patterns can be very complex as well. The approach weaves the synchronization code into the computation code.

Another approach for making correct programs is to check their correctness. Model checkers [12], [21], [125] do this task automatically. Model checkers get a specification and a finite model of the program as input and check whether the model corresponds to the specification and if not, they give a counter-example. The specification is usually a temporal logic specification, and model checkers use a tableau method. This way the problem of the state explosion occurs in the case of using model checkers, that is, the models of the systems grow exponentially in the number of the components of the system. In [27] there is a partial solution for the state explosion problem by using only similar components. This method avoids building a full state transition graph and because of the similarity, only a transition graph of two components has to be built. An other solution is presented in [93], which solves the problem of the state explosion by using a special method, a so called symbolic model checking. The method avoids building a state transition graph by using boolean formulae to represent sets and relations.

The temporal logic formulae are usually too difficult and hard to understand, so developers do not like to use them. In [9] a set of syntactic sugar is introduced to make the using of CTL [38] more usable.

A solution can be found in [129] for synthesizing the synchronization code of general pipeline systems. There are many problems in the case of pipeline systems, especially synthesizing the synchronization code of the two ends of the pipeline. That thesis contains an approach for splitting the system to parts, processes of which are similar and after that a simple method can be used for generating the synchronization between groups.

Chapter 6

Summary and conclusions

We have given an algorithm for specifying the synchronization of object-oriented systems. We have distinguished two cases: in the first case parallel objects are working and using resources while communicating with each other directly. In the second case passive shared classes are used, where the shared class provides the resources. The second case is traced back to the first case.

In our algorithm, users have to use more tools to be able to run the synthesis algorithm. First, UML class diagrams have to be made. By planning class diagrams, designers have to take care of determining the different synchronization classes. The synchronization class diagram can be separated from the other parts, and the two kinds of class diagrams can be joined with associations. This can help to avoid the state explosion problem.

Users have to determine the states of the types (classes). The state sets of the classes are joined during our method. After generating the abstract code, the resulted synchronization skeletons can be split based on the state sets of the classes, therefore a separate synchronization code is generated for all classes.

Inheritance can cause problems. There are cases when it is desired to use an already completed class and its synchronization code. We solved the problem with introducing state mappings in a special case, when equivalent states can be found in the ascendant class. Unfortunately, we could not solve the problem generally, but we have given an algorithm that can be used if we cannot solve the problem (in this case refactoring [111], [137] is

needed and dynamic binding cannot be used – the full synchronization code have to be regenerated and the old one cannot be used, but it can be done automatically).

We have given a way to join the synchronization code to the computation code. This can be done through UML state diagrams. In fact, synchronization states represent states of the computation (they always depend on the computation, because the computation has to be synchronized). When using this approach, some computation states have to be labelled with synchronization states (joining the two kinds of states). Special stereotypes are used for denoting the synchronization transactions. Based on this diagram, synchronization calls can be weaved into the computation code. We have given the possibility of changing the synchronization behaviour. This can be specified in the state diagrams as well.

The synchronization specification is made in MPCTL*. But for the sake of the usability, we introduced a graphical representation of MPCTL* and extended the base primitive elements with new abstract constructions so we can express quite difficult synchronization behaviours easily.

We have introduced a method for specifying the interconnection relation by using first order predicate calculus with spatial operators. The original spatial operators have been introduced in [7] but we extended them with two parameters: two types of the connected objects.

After the developer has made the class diagram (with a separated synchronization structure), specified the states with their special properties (e.g. state mapping, etc.), planned the connection of the synchronization and the computation by using the state diagrams of the synchronizable computation classes, and defined the interconnection relation, the method of Attie and Emerson can be run, which results in a finite non deterministic automaton. The generated automaton can be separated to the individual classes based on the reachable state sets of the types.

We have given an implementation method of the synchronization code. This method generates several classes, but these classes are fully generated. Furthermore, the largest part of the generated code is constant, so it can be optimized effectively. In fact, the only non constant methods are method

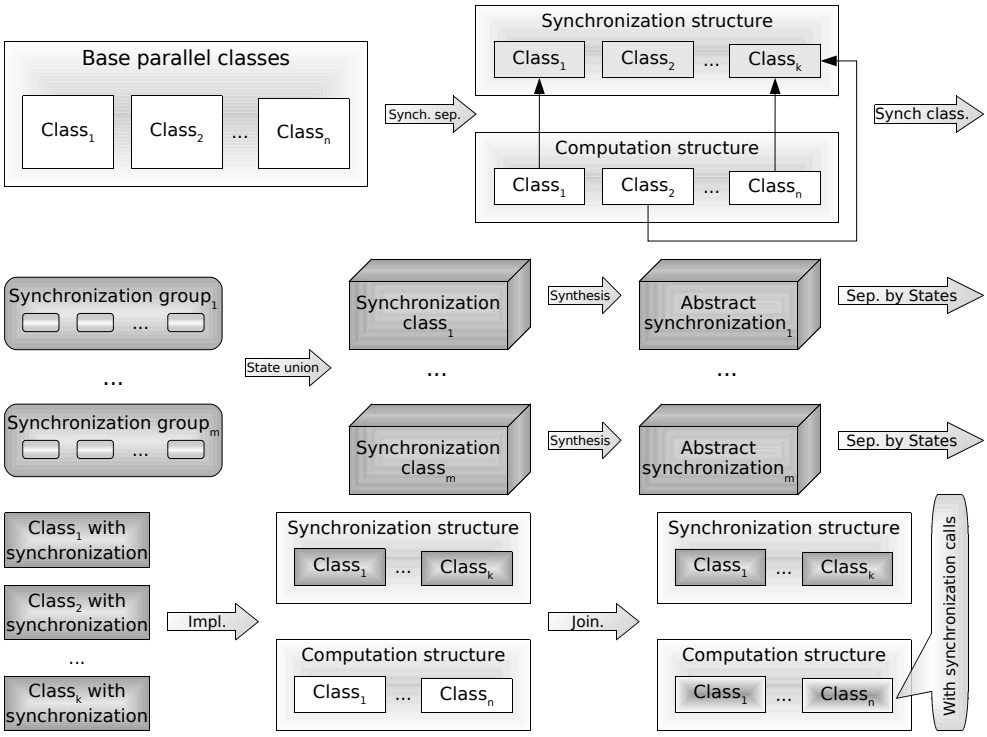

Figure 6.1: Functioning of the synthesis process. State mappings have to be handled by generating classes with synchronization codes.

setState, which controls the synchronization (but this method also contains constant parts) and methods *checkConsistency* and *checkI*, which check the consistency of the interconnection relation. Furthermore, synchronization calls are generated into the computation code in the proper places, so after the execution of the generation method, users do not need to take care of the synchronization, they can concentrate to the computation.

We have decided to use semaphores and shared variables to implement the synchronization and communication between objects. We use this technique because it is very general and widely reachable. If the processes run in different places, then shared memory blocks cannot be used. But using shared classes for synchronizing the active objects is a workable solution. Figure 6.1 shows the synchronization process.

We have not given a tool that implements the introduced algorithm, we only have given the steps of the synthesis method. But we constructed our

algorithm to be language independent. Of course it is not fully language independent, we only concentrated to object-oriented languages that are used in the practice. We developed the method for Java, but we did it so that it can be easily transformed to an other language.

This method has limits. We improved the handling of the state explosion problem but we have not solved it. As we realized, several systems have a synchronization where our method can be used. Unfortunately, there are systems, where too many states are generated to handle. But if we follow the recommendation of an extension of the object-oriented paradigm, and passive shared objects are used to control the synchronization [68], then the problems are kept handleable (because the synchronization of shared classes can usually be traced back to simple parallel object-oriented systems). However, if there are lots of objects in the system with lots of connections, then running of the generated synchronization code becomes too slow.

Our method consists of several steps and generates several new classes. But developers need to use only the interface *SynthesisObject* and the other types are hidden. If the system has to be used in a distributed way, then class *SharedObject* is put in a central place to control the connections and the condition evaluating, so developers need to use this class as well – we have not considered this possibility yet (see Chapter 7). Besides, the design of the synchronization of a system is not so easy yet, even with the introduced methods, which are built on popular (UML based) techniques, because the algorithm consists of too many steps.

We cannot express everything in the used model. These kind of temporal logic specifications can be mainly used to describe progress features and so it can be expressed easily that some statements of objects have to be executed in mutual exclusive mode. Furthermore, we can describe features of the connected processes, but other properties can be expressed very hard, e.g., if we want to describe that at most ten objects can be in a critical section, that can be solved very hard and makes the system too difficult.

Chapter 7

Future Work

We have given an algorithm but we do not have a tool that implements it. Since the skeleton of the computation code can be generated by using a CASE tool [104], [108], etc., the desired implementation is to embed our algorithm in one of them. We have given an implementation in [59] for generating the XML [33], [146] representation of the MPCTL* formulae from the synchronization diagram – it has to be extended because new elements of the synchronization diagrams were introduced. After that we could generate the synchronization skeleton – which is an automaton – by using an existing implementation of a tableau algorithm and store the result in an XML format. Based on this representation the concrete program code could be generated by following the steps of our algorithm – the (large) constant parts of the code can be stored in regular files and these only have to be copied. The synchronization calls could be generated in the skeleton of the computation that was generated by the given CASE tool. Our goal in the future is to implement these steps.

We have given a solution for generating the synchronization code of object-oriented systems, but we have not proven the correctness of the algorithm. The correctness of the constant codes could be proven by defining the semantics of the methods with pre- and postconditions and using the Owicki-Gries method [105]. Furthermore, it can be proven with this method, that only one object can be in the phase where it evaluates the conditions of the state transitions. Unfortunately, some methods are too large to handle them with this algorithm (correctness includes the freedom from deadlocks and

starvation – only for token capturing and reaching of I). Besides, we have to prove the correctness of the transformation of the condition evaluation and the concrete implementation of the abstract automatons.

Parallel programs are very complex, so designers need tools that can be used for proving some important properties. These properties are freedom from deadlock or starvation, other liveness properties, etc. Attie and Emerson developed algorithms to check these properties, but because of the dynamic interconnection relation those algorithms are not working. We have to give methods to check the mentioned properties in order that safer programs can be made.

As we mentioned before, we have not solved the problem of the state explosion, we only reduced it. We need to find other solutions that help designers to generate the synchronization. One of the possibilities is to find an algorithm that merges states of different classes. With such an algorithm, the problem could be made smaller because the large number of states could be reduced further. In fact, such an algorithm could be used for reducing the synchronization specification, so it could be used after planning the synchronization – therefore, it cannot help in the design phase – but it could be used to make the system simpler and this can give hand with maintaining the system.

We have given a graphical solution for designing the synchronization code. Some new abstract elements were introduced to make the design simpler. We have to analyse lots of concurrent systems in order to provide new elements for the synchronization diagrams to make designing the synchronization of systems easier.

The interconnection relation can be specified by using logical formulae and extended spatial operators. As it can be seen in Section 4.1.3, specification of I can be very complex. It would be better to give a simpler – perhaps graphical – method for specifying I.

There can be difficulties by reusing the synchronization code of a class. We have given a solution for the state partitioning inheritance anomaly only in special cases. We have to examine what other problems can appear and what we can do to avoid them. It is very important of reusing the code

without problems in order to use the tools of object-orientedness.

A possible solution for the better reuse of the synchronization is to use several setState operations instead of a general one. So if a part of a function cannot be reused, the other functions are kept and only a small part has to be regenerated. We have to examine this approach because it has many disadvantages, e.g. if we separate the functions along the state transitions, dynamic values are transformed to constant names. We have not succeeded to get a usable solution yet.

The generated code is usually not optimal. You can find a solution for optimizing the generated code in [129], which can be adopted in our case. But optimization is a very hard task because it depends on the interconnection relation and if the interconnection relation is dynamic, then other problems are arisen. We have to find better solutions.

We designed the structure of the synchronization code so that class *SharedObject* can handle the interconnection of the objects and it controls the condition evaluating as well. Furthermore, token informations are stored in *SharedObject* so it cannot cause problems if an object is unreachable. This is the first step to examine the possibilities of running the generated program code in a distributed environment. Of course there can be problems, e.g. if the central *SharedObject* cannot be reached, etc. We have to examine our possibilities and the problems that can be arisen and we have to find some usable solutions.

It would be desired to find a language independent representation and the last step of code generation would be compiling to concrete code from this representation. It can be seen that if we choose this way, we have to find a representation that is expressive enough and its translation to other languages is simpler than the original problem. An expressive XML [33], [58], [82], [139], [146] representation seems to be appropriate for this purpose, but we still have to work on this problem.

Bibliography

[1] B. Albahari, P. Drayton, B. Merrill: *C# Essentials*, O'Reilly Media, Inc. (2002), ISBN 059 600 315 3.

[2] P. America: *Inheritance and Subtyping in a Parallel Object-oriented Language*, LNCS Vol. 276 (1987), pp. 234-242.

[3] P. America: *POOL-T: a parallel object-oriented language*, In Object-oriented concurrent systems (Yonezawa A. and Tokoro M., eds), MIT Press (1987), pp. 199-220.

[4] G. R. Andrews: *A Method for Solving Synchronization Problems*, Science of Computer Programming 13 (1989/90), pp.1-21.

[5] P. C. Attie, A. Arora, E. A. Emerson: *Synthesis of fault-tolerant concurrent programs*, ACM TOPLAS Vol. 26, Issue 1 (2004), pp. 125-185.

[6] P. C. Attie, E. A. Emerson: *Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation*, ACM TOPLAS, Vol. 23, No. 2 (2001).

[7] P. C. Attie, E. A. Emerson: *Synthesis of Concurrent Systems with Many Similar Processes*, ACM TOPLAS Vol. 20, No. 1 (1998), pp. 51-115.

[8] M. Awad et al.: *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice Hall, Englewood Cliffs (1996).

[9] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, Y. Rodeh: *The Temporal Logic Sugar*, Lecture Notes in Computer Science, Vol. 2102 (2001), pp363 - 367.

[10] F. Belik, L. Kozma, D. Krznaric: *Avoiding Deadlock and Starvation in Distributed Resource Allocation System*, Proc. of the 13th International Conference of Parallel and Distributed Systems (1993), pp. 112-117.

[11] M. Ben-Ari, Z. Manna, A. Pnueli: *The temporal logic of branching time*, Annual Symposium on Principles of Programming Languages archive, Proceedings

of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York (1981), pp. 164-176.

[12] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer (2001), ISBN 354 041 523 8.

[13] P. A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley 1987, ISBN 020 110 715 5.

[14] R. Binder: *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley (2000).

[15] G. Booch: *UML in Action*, Comm. of the ACM Vol. 42. No. 10, (1999).

[16] J-P. Briot, A. Yonezawa: *Inheritance and Synchronization in Concurrent OOP*, LNCS Vol. 276 (1987), pp. 32-40.

[17] H. K. Browne: *Adding Concurrency to LOOM*, WILLIAMS COLLEGE, Williamstown, Massachusetts (1997).

[18] A. Burns, G. Davies: *Concurrent Programming*, Addison-Wesley Pub. Co. Wokingham England (1993), ISBN 020 154 417 2.

[19] M. Cantù: *Mastering Borland Delphi 2005*, Sybex Inc. (2005), ISBN 078 214 342 3.

[20] D. Chappell: *Understanding .NET (2nd Edition) (Independent Technology Guides)*, Addison-Wesley (2006), ISBN 032 119 404 7.

[21] E. M. Clarke Jr., O. Grumberg, P. A. Peled: *Model Checking*, The MIT Press, Cambridge (1999), ISBN 026 203 270 8.

[22] P. Coad, E. Yourdon: *Object-Oriented Analysis*, Yourdon Press/Prentice Hall, Englewood Cliffs (1990).

[23] D. Coleman et al.: *Object-Oriented Development: The FUSION Method*, Prentice Hall, Englewood Cliffs (1994).

[24] S. Cook, J. Daniels: *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, Englewood Cliffs (1994).

[25] J. W. Cooper: *Java Design Patterns: A Tutorial*, Addison-Wesley (2000), ISBN 020 148 539 7.

[26] L. Crnogorac, A.S. Rao and K. Ramamohanarao: *Inheritance Anomaly – A formal Treatment*, In Proc. of FMOODS'97, Chapman & Hall. (1997), pp. 319-334.

[27] Á. Dávid, L. Kozma, T. Pozsgai: *On the model checking of a system consisting of many similar components*, Annales Univ. Sci. Budapest. (2007) – under publishing.

[28] S. Demeyer, S. Ducasse, O. Nierstrasz: *Object Oriented Reengineering Patterns*, Morgan Kaufmann (2002), ISBN 155 860 639 4.

[29] Dia
`http://www.gnome.org/projects/dia`

[30] E. W. Dijkstra: *Cooperating sequential processes*, In Programming Languages, F. Genuys, Ed, Academic Press, New York (1968), pp. 43-112.

[31] L. K. Dillon et al.: *A Graphical Interval Logic for Specifying Concurrent Systems*, ACM Trans. on Soft. Eng. and Methodology, 3 (2) (1994), pp. 131-165.

[32] P. Diviánszky, R. Szabó-Nacsa, Z. Horváth: *Refactoring via Database Representation*, In Csőke et al. eds.: Proceedings of 6th International Conference on Applied Informatics, Eger, Hungary, Vol. I. (2004), pp. 129-136.

[33] R. Eckstein: *XML Pocket Reference* (1999), ISBN 156 592 709 5.

[34] M. Dwyer, J. Hatcliff, M. Mizuno, M. Neilsen, G. Singh: *Automatic Derivation, Integration and Verification of Synchronization Aspects in Object-Oriented Design Methods* Report for DARPA Order K203/AFRL Contract F33615-00-C-3044 (2001).

[35] H. El-Rewini, T. G. Lewis, H. H. Ali: *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall (1994), ISBN 013 099 235 6.

[36] A. Ebnenasir, S. S. Kulkarni: *Automatic addition of liveness.* Technical Report MSU-CSE-04-22, Department of Computer Science, Michigan State University, East Lansing, Michigan (2004).

[37] E. A. Emerson: *Temporal and modal logic*, Handbook of theoretical computer science, Vol. B, Formal models and semantics, MIT Press, Cambridge (1991), pp. 995-1072.

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

[39] E. A. Emerson, J. Srinivasan: *A decidable temporal logic to reason about many processes*, In Proc. of the ninth annual ACM symposium on Principles of distributed computing, ACM, New York (1990), pp. 233-246.

[40] F. E. Fich: *The complexity of computation on the Parallel Random Access Machine*, In Synthesis of Parallel Algorithms, J.H. Reif (ed.) (1993), pp. 843-899.

[41] M. Fowler: *A UML Testing Framework*, Software Development, Vol. 7, Issue 4 (1999), pp. 41-46.

[42] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995), ISBN 020 163 361 2.

[43] C. Girault, R. Valk: *Petri Nets for Systems Engineering*, Springer (2007), ISBN 354 041 217 4.

[44] J. Gosling, K. Arnold: *The Java Programming Language*, Addison-Wesley (1994).

[45] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*, Addison-Wesley (1996).

[46] J. Gosling, F. Yellin: *The Java Application Programming Interface, Volume 1-2*, Addison-Wesley (1996).

[47] I. Graham: *Migrating to Object Technology, Addison-Wesley*, Reading MA (1994).

[48] R. Grimes: *Professional Dcom Programming*, Peer Information Inc. (1997), ISBN 186 100 060 X.

[49] H. G. Gross: *Component-Based Software Testing with UML*, Springer (1998), ISBN 354 020 864 X.

[50] A. Gupta: *Analysis and Design of Scalable Parallel Algorithms for Scientific Computing*, PhD thesis, University of Minnesota, Minneapolis, MN (1995).

[51] J. V. Guttag: *Abstract Data Types and Development of Data Structures*, Comm. ACM 20, 6 (1977), pp. 396-404.

[52] Sz. Hajdara: *Objektum-elvű rendszerek szintézisproblémái*, Professional conference of BME, Department of Measurement and Information Systems (2007).

[53] Sz. Hajdara, L. Kozma, B. Ugron: *Párhuzamos programok szintézise és objektum elvű kiterjesztése*, V. Országos Objektum-orientált Konferencia, Dobogókő (2002).

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

[55] Sz. Hajdara, B. Ugron: *Analyzing the Effectiveness of Synthesized Synchronization Codes*, In: Proc. of 6th Joint Conference on Mathematics and Computer Science, Pécs (2006), pp. 38.

[56] Sz. Hajdara, B. Ugron: *An example of generating the synchronization code of a system composed by many similar objects*, In: Proc. 17th European Conference on Object-Oriented Programming (ECOOP), The 13th Workshop for PhD Students in Object-Oriented Systems, LNCS 3013 (2003), p. 55.

[57] Sz. Hajdara, B. Ugron: *An extension of synthesis of the synchronization of object-oriented pipeline systems*, Pure Mathematics and Applicatons, Vol. 15, No. 2-3 (2004), pp. 157-169.

[58] Sz. Hajdara, B. Ugron: *Describing Semantics of Data Types in XML*, Conference brochure: 6th International Conference on Applied Informatics, Eger, Vol. 1 (2004) pp. 173-180.

[59] Sz. Hajdara, B. Ugron, L. Kozma: *A graphical interface for specifying the synchronization of a concurrent system*, Periodica Politechnica, Budapest (2005) – submitted.

[60] D. Harkey, R. Orfali: *Client/Server Programming with Java and CORBA, 2nd Edition*, John Wiley & Sons (1998), ISBN 047 124 578 X.

[61] P. Harmon, C. Hall: *Intelligent Software Systems Development: An IS Manager's Guide*, Wiley (2001), ISBN 047 159 244 7.

[62] M. Henning, S. Vinoski: *Advanced CORBA programming with C++*, Addison-Wesley (1999), ISBN 020 137 927 9.

[63] C. A. R. Hoare. *Monitor: an operating system structuring concept*, Communications of the ACM, 17(10), (October 1974) pp. 549-557.

[64] C. S. Horstman, G. Cornell: *Core JAVA 1.2 Volume I - Fundamentals*, Sun Microsystem-Prentice Hall (1999).

[65] Z. Horváth: *The Weakest Precondition and the Specification of Parallel Programs*, In: Proc. of the Third Symposium on Programming Languages and Software tools, Kaariku, Estonia (1993), pp. 24-34.

[66] A. Housni, M. Trehel: *Distributed Mutual Exclusion Token-Permission Based by Prioritized Groups*, AICCSA, In. Proceedings of the ACS/IEEE International Conference on Computer Systems and Appl, IEEE Computer Society, (2001) p. 253.

[67] I. Jacobson et al.: *Object-Oriented Software Engineering: A Use Case Driven Approach*, New York, Addison-Wesley (1992).

[68] S. Jagadish, R. K. Shyamasundar: *UML based approach to specify secured, fine-grained concurrent acces to shared variables*, Journal of Object Technology, Vol. 6, No. 1 (2007), pp. 107-119

[69] J. Kerievsky: *Refactoring to Patterns*, Addison-Wesley (2004), ISBN 032 121 335 1.

[70] L. Kozma: *Absztrakt adattípusok párhuzamos környezetben*, Kandidátusi értekezés (1982).

[71] L. Kozma: *On the starvation problem of concurrent programs*, PU.M.A., Vol. 15, No. 2-3 (2004), pp. 203-212.

[72] L. Kozma: *Proving the Correctness of Implementations of Shared Data Abstractions*, LNCS Vol. 137 (1982), pp. 227-241.

[73] L. Kozma, L. Varga: *A szoftvertechnológia elméleti kérdései*, ELTE Eötvös Kiadó (2003).

[74] P. Kruchten: *The Rational Unified Process*, Addison-Wesley (2003), ISBN 032 119 770 4.

[75] C. P. Kruskal, L. Rudolph, M. Snir: *A complexity theory of ecient parallel algorithms*, Theoret. Comput. Sci., Vol. 71, No. 1 (1990), pp. 95-132.

[76] S. S. Kulkarni, A. Ebnenasir: *Automated synthesis of multitolerance*, In International Conference on Dependable Systems and Networks (2004) pp. 209-219.

[77] V. P. Kumar and A. Gupta: *Analysing Scalability of Parallel Algorithms and Architectures*, Journal of Parallel and Distributed Computing, Vol. 22, No. 3 (1994), pp. 379-391.

[78] R. Kurki-Suonio, Tommi Mikkonen: *Liberating Object-Oriented Modelling from Programming-Level Abstractions*, ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modelling Techniques Ed. by Haim Kilov & Bernhard Rumpe, Technishe Universität München, (1997), pp. 115-121.

[79] G. Kutty, Y. S. Ramakrishna, L. E. Moser, L. K. Dillon, P. M. Melliar-Smith: *A Graphical Interval Logic Toolset for Verifying Concurrent Systems*, In. Proc. of the 5th Computer Aided Verification, Springer-Verlag, London (1993), pp. 138-153.

[80] R. Laddad: *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications (2003), ISBN: 193 011 093 6.

[81] A. van Lamsweerde, M. Sintzoff: *Formal Derivation of Strongly Correct Concurrent Programs*, Acta Informatica, Vol. 12, Fasc. 1 (1979), pp. 1-31.

[82] S. St. Laurent: *XML: Elements of Style*, New York (2000), ISBN 007 212 220 X.

[83] F. T. Leighton: *Introduction to Parallel Algorithms and Architectures* (1992).

[84] D. Liang: *Introduction to Java Programming-Comprehensive Version (6th Edition)*, Prentice Hall (2006), ISBN 013 222 158 6.

[85] B. Lisper: *Synthesizing Synchronous System by Static Scheduling in Space-Time*, LNCS Vol. 362 (1987), ISBN 038 751 156 3.

[86] T. Love: *Object Lessons: Lessons Learned in Object-Oriented Development Projects*, SIGS Books, Inc., New York (1993).

[87] Z. Manna, R. Waldinger: *The automatic synthesis of recursive programs*, SIGPLAN Notices, Vol. 12, No. 8 (1977), pp. 29-36.

[88] Z. Manna, R. Waldinger: *The synthesis of structure-changing program*, In: Proc. 3rd International Conference on Software Engineering, Atlanta, USA (1978), pp. 175-178.

[89] Z. Manna, P. Wolper: *Synthesis of Communicating Processes from Temporal Logic Specifications*, ACM TOPLAS 6 (1984) pp. 68-93.

[90] C. Marshall: *Enterprise Modeling With Uml*, Addison-Wesley (1999), ISBN 020 143 313 3.

[91] J. Martin, J. Odell: *Object-Oriented Methods: A Foundation*, Prentice Hall, Englewood Cliffs (1995).

[92] S. Matsuoka and A. Yonezawa: *Analysis of inheritance anomaly in object-oriented concurrent programming languages.* In G. Agha, P. Wegner, and A. Yonezawa, editors, Research Directions in Concurrent Object-Oriented Programming, MIT Press (1993), pp 107-150.

[93] K. L. McMillan: *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic, 1993.

[94] B. Meyer: *Object-Oriented Software Construction Second Edition*, Prentice Hall (1997), ISBN 013 629 155 4.

[95] B. Meyer: *Systematic Concurrent Object-Oriented Programming*, Communications of the ACM, Vol. 36, No. 9 (1993), pp. 56-80.

[96] G. Milicia, V. Sassone: *The inheritance anomaly: ten years after*, In Proc. of the 2004 ACM symposium on Applied computing, ACM Press (2004) pp. 1267-1274.

[97] T. Minoura, G. Wiederhold: *Resilient Extended True-Copy Token Scheme for a Distributed Database System*, IEEE Trans. Software Eng. 8(3) (1982), pp. 173-189.

[98] M. Mizuno, G. Singh, M. Nielsen: *A structured approach to develop concurrent programs in UML*, In Proc. of third International Conference on UML, LNCS 1939, Springer (2000), pp. 541-565.

[99] K. H. Mortensen: *Automatic Code Generation from Coloured Petri Nets for an Access Control System*, In Proc. of the Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark, Jensen, K., Editor (1999), pp. 41-58.

[100] R. H. B. Netzer and B. P. Miller: *On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions*, Proceedings of the 1990 International Conference on Parallel Processing, Vol. II (1990) pp. 93-97.

[101] J. A. Nikara, J. H. Takala, J. T. Astola: *Discrete cosine and sine transforms: regular algorithms and pipeline architectures*, Signal Processing Vol. 86, Issue 2 (2006) pp. 230-249.

[102] G. J. Nyékyné et al. ed.: *Java 2 útikalauz programozóknak 1.3 (Programmers' Guide to Java 2, version 1.3)*, ELTE TTK Hallgatói Alapítvány, Budapest, Hungary (2001).

[103] G. J. Nyékyné et al. ed.: *J2EE útikalauz programozóknak (Programmers' guide to J2EE)*, Kiskapu Kft., Budapest, Hungary (2002).

[104] P. W. Oman, A. J. Bowles, R. Mount, G. Karam, D. Kalinsky, M. Tervonen, V. Bundonis, H. Fischer, M. Fish, D. Longshore, N. Akiha: *CASE: analysis and design tools*, Software, IEEE, Vol. 7, No. 3 (1990), pp. 37-43.

[105] S. Owicki and D. Gries: *An Axiomatic Proof Technique for Parallel Programs*, Acta Informatica 6 (1976), pp. 319-340.

[106] A. Pataricza: Formális módszerek az informatikában, Typotex (2006), ISBN 963 954 890 1.

[107] A. Pope: The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture, Addison Wesley (1997), ISBN 020 163 386 8.

[108] T. Quatrani: *Visual Modeling with Rational Rose 2002 and UML*, Addison-Wesley (2003), ISBN 020 172 932 6.

[109] A. A. Radenski: *Object-Oriented Programming and Parallelism: Introduction*, Information Sciences, Vol. 93, No. 1 (1996), pp. 1-7.

[110] C. V. Ramamoorthy, H. F. Li: *Pipeline Architecture*, ACM Computing Surveys (CSUR), Vol. 9, Issue 1 (1977), pp. 61-102.

[111] Refactoring
`http://www.refactoring.com`

[112] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object Oriented Modelling and Design*, Prentice Hall (1991).

[113] J. Ryser, M. Glinz: *A practical approach to validate and testing software systems using scenarios*, In Quality Week, Brussel (1999).

[114] S. Sahni and V. Thanvantri: *Parallel computing: Performance metrics and models*, Research Report, Computer Science Department, University of Florida (1995).

[115] B. Selic, G. Gullecson, P. Ward: *Real-Time Object-Oriented Modeling*, Wiley, New York (1994).

[116] R. Sessions: *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons, Inc. (1997), ISBN 047 119 381 X.

[117] A. U. Shankar: *An Introduction to Assertional Reasoning for Concurrent Systems*, ACM CSUR, Vol. 25, Issue 3 (1993), pp. 225-262.

[118] S. Sike, L. Varga: *Objektum elvű modellalkotás UML-ben – Példatár definíciókkal* –, ELTE, TTK, Informatikai Tanszékcsoport, Budapest (2001).

[119] S. Sike, L. Varga: *Szoftvertechnológia és UML*, ELTE Eötvös Kiadó (2001), ISBN 963 463 477 X.

[120] O. Sinnen: *Task Scheduling for Parallel Systems*, Wiley-Interscience (2007), ISBN 047 173 576 0.

[121] R. M. Smullyan: *First Order Logic*, Springer Verlag, New York (1971).

[122] E. W. Stark: *Semaphore primitives and starvation-free mutual exclusion*,J. ACM, Vol. 29, No. 4, (1982), pp. 1049-5411.

[123] B. Stroustrup: *The C++ Programming Language*, Addison-Wesley (2000), ISBN 020 170 073 5.

[124] K. C. Tai: *Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs*, In: Proc. Parallel Processing, ICPP 1994 International Conference, Vol. 2 (1994), pp. 69-72.

[125] N. T. Thang, T. Katayama: *Specification and verification of intercomponent constraints in CTL*, In Proc. of the 2005 conference on Specification and verification of component-based systems, Vol. 31, Issue 2 (2005), pp. 1-8.

[126] TIOBE Software
`http://www.tiobe.com/tpci.htm`

[127] C. Tomlinson, V. Singh: *Inheritance and synchronization with enabled-sets*, ACM SIGPLAN Notices, Vol. 24 No. 10 (1989), pp. 103-112.

[128] D. C. Toward: *A Method of Object-oriented Concurrent Programming*, Comm. of the ACM, Vol. 36, No. 9 (1993), pp. 90-102.

[129] B. Ugron: *Synthesis of the synchronization code of pipeline systems*, PhD Thesis, ELTE IK, Budapest (2007).

[130] B. Ugron, Sz. Hajdara: *Synthesis of the synchronization of pipeline systems*, Conference brochure: 6th International Conference on Applied Informatics, Eger, Vol. 2 (2004) pp. 351-359.

[131] B. Ugron, Sz. Hajdara: *Synthesis of the synchronization of general pipeline systems*, Acta Cibernetica 17 (2004), pp. 123-151.

[132] B. Ugron, L. Kozma, Hajdara, Sz.: *A strukturált programozástól az objektum elvű technológiáig*, V. Országos Objektum-orientált Konferencia, Dobogókő (2002).

[133] B. Ugron, L. Kozma, Sz. Hajdara, L. Blum: *Implementations of synchronization of concurrent objects*, Annales Univ. Sci. Budapest., Sect. Comp. 23 (2004), pp. 79-102.

[134] J. D. Ullman, J. Widom: *A First Course in Database Systems*, New Jersey (1997).

[135] UML – Unified Modeling language
`http://www.omg.org/technology/documents/formal/uml.htm`

[136] Cs. Vég, I. Juhász: *Objektumorientált világ*, IQSOFT Rt. (1998).

[137] W. C. Wake: *Refactoring Workbook*, Addison-Wesley (2003), ISBN 0321 10 929 5.

[138] K. Waldén, J-M. Nerson: *Seamless Object-oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice Hall, Englewood Cliffs (1995).

[139] P. Walmsley: *Definitive XML Schema* (2001), ISBN 013 065 567 8.

[140] P. Wegner: *Classification in Object-Oriented Systems*, SIGPLAN Notices, Vol. 21, No. 10 (1986), pp. 173-183.

[141] G. Wiederhold, X. Qian: *Modeling Asynchrony in Distributed Databases*, In: Proc. of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA, IEEE Computer Society (1987), pp. 246-250.

[142] C. P. Willis: *Analysis of inheritance and multiple inheritance*, Software Engineering Journal (1996), pp. 215-224.

[143] P. Wolper: *The Tableau Method for Temporal Logic: an Overview*, Logique at Anal. 28 (1985), pp. 119-136.

[144] A. S. Woodhull, A. S. Tanenbaum: *Operating Systems. Design and Implementation.*, Prentice Hall (1997)

[145] X. Wu: *Performance Evaluation, Prediction and Visualization of Parallel Systems* (1999), ISBN 079 238 462 8.

[146] XML – Extensible Markup Language
`http://www.w3.org/XML`

Appendix A

We show an example for generating the synchronization code.

This example is a simulation of a surgery. There are normal patients that can be in normal state (N), in the waiting room (W) and in the examination room (E) and the ordered sequence of the states must be $N \rightarrow W \rightarrow E$. If a patient is in state E then other patients cannot be in this state. Ther are severely ills, they can be in states N, W and in a new state, severe diseased in the examination room (S) and they can always step into S from W (and the ordered sequence of their states is $N \rightarrow W \rightarrow S$). If a severe diseased patient is in the examination room, it does not exclude a normal patient because different doctors examine severe and normal ills.

In the following we solve this problem. We generate the synchronization specification and we show a computation code that uses the synchronization and synchronization calls are generated in the computation code.

The structure of the system

The class diagram of the system is shown in Figure 7.1.

The synchronization code is separated from the computation code and class *Human* contains the computation code. We assume that class *Human* represents a human type and humans can be sick. If a human is sick, it goes to the hospital. In the hospital a "triage nurse" examines her and nurse can decide whether her disease is severe. If it is so, triage nurse makes her synchronization behaviour to severe, else her synchronization is going to be normal. (We assume that a triage nurse can handle several patients in a time – because her job is only to take a decision). That is the reason why

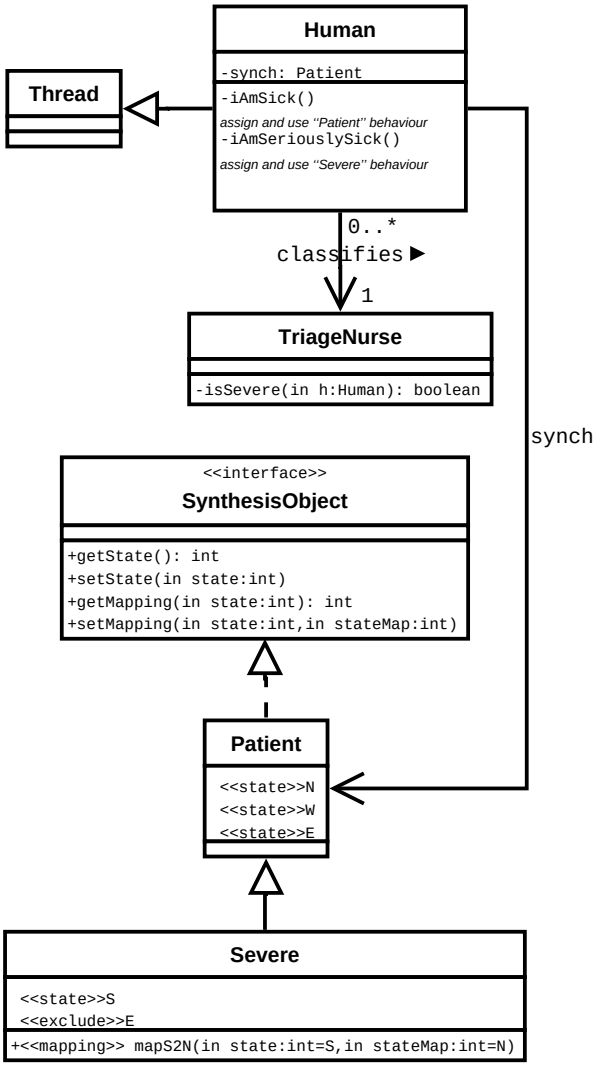

Figure 7.1: Class diagram of the surgery simulation system.

triage nurses do not get any synchronization. In fact, this means that several patients can be in the "triage room" where the triage nurse is working. After the nurse has decided, severe diseased patients can go into the examination room immediately in order to be examined (by several doctors so examination of them can fulfilled in parallel), while normal patients can step into the examination room only if no other normal patients are in there because it is assumed that only one doctor examines the normal patients.

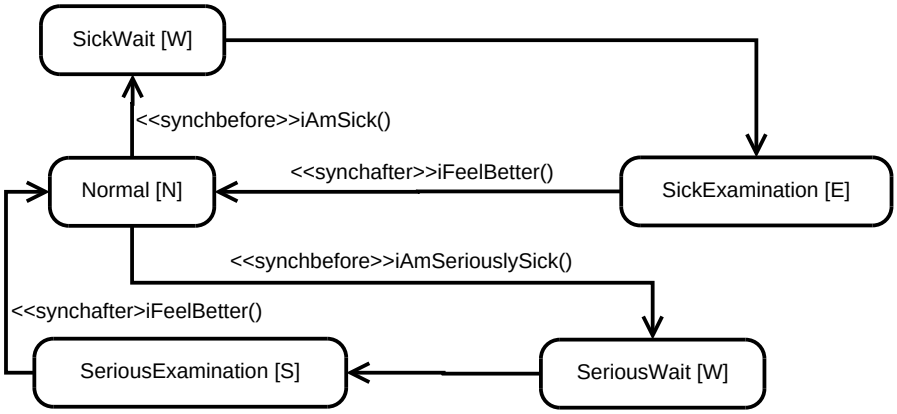

Figure 7.2: State diagram of class Human.

The state sets

The state sets are the following:

- State set of class *Patient* = {N, W, E}
- State set of class *Severe* = {N, W, S}

Unioning the sets of states means that the states of entity O_i are in set $\{N, W, E, S\}$ (the appropriate atomic propositions are N_i, W_i, E_i, S_i – N_i means that object with reference i is in state N).

Connecting the computation code to the synchronization code

The synchronization code can be connected to the computation code through the state diagram of class *Human*, which can be seen in Figure 7.2.

The interconnection relation

In our example, every object is connected to all the others because patients affect each other. So the specification of I is the following:

$$(\uparrow, \uparrow) \bigwedge_{ij} true.$$

The specifictaion of the synchronization

The temporal logic specification of the system is the following:

$$\bigwedge_i N_i$$

$$\bigwedge_i AG(N_i \Rightarrow (AY_i W_i \wedge EX_i W_i))$$

$$\bigwedge_i AG(W_i \Rightarrow (AY_i(E_i \vee S_i)))$$

$$\bigwedge_i AG(E_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

$$\bigwedge_i AG(S_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

$$\bigwedge_i AG(W_i \Rightarrow (EX_i S_i \vee AF E_i))$$

$$\bigwedge_i AG(N_i \equiv \neg(W_i \vee E_i \vee S_i))$$

$$\bigwedge_i AG(W_i \equiv \neg(N_i \vee E_i \vee S_i))$$

$$\bigwedge_i AG(E_i \equiv \neg(N_i \vee W_i \vee S_i))$$

$$\bigwedge_i AG(S_i \equiv \neg(N_i \vee W_i \vee E_i))$$

$$\bigwedge_{ij} AG((N_i \Rightarrow AY_j N_i) \wedge (N_j \Rightarrow AY_i N_j))$$

$$\bigwedge_{ij} AG((W_i \Rightarrow AY_j W_i) \wedge (W_j \Rightarrow AY_i W_j))$$

$$\bigwedge_{ij} AG((E_i \Rightarrow AY_j E_i) \wedge (E_j \Rightarrow AY_i E_j))$$

$$\bigwedge_{ij} AG((S_i \Rightarrow AY_j S_i) \wedge (S_j \Rightarrow AY_i S_j))$$

$$\bigwedge_{ij} AG(\neg(E_i \wedge E_j))$$

The generated code

Based on the above, following code can be generated. Of course all implementation of Human cannot be generated but we show an example implementation of this class in order to visualize, how to use it.

We remark that in this example we implemented *setState* in the simpler way, i.e. token collecting can be made in exclusive mode. Furthermore, function *makeI* uses ostrich politics, that is, implementation assumes that this function is used correctly, so no writing I is possible in a state, which is different from the initial. Because there are no quantifiers in the specification of I, *makeI* does not has to examine other objects after modifying I.

```
// Semaphore.java
// --------------

public class Semaphore {
    private int count;
```

```
    public Semaphore(int initialCount) {
        count = initialCount;
    }

    public synchronized void P() {
        while ( count <= 0 ) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        --count;
    }

    public synchronized void V() {
        ++count;
        notify();
    }
}

// ---------------------------------------------------------

// InvalidStateTransitionException.java
// ------------------------------------

public class InvalidStateTransitionException
        extends Exception {
    private int stateFrom;
    private int stateTo;

    public InvalidStateTransitionException(int stateFrom,
            int stateTo) {
        this.stateFrom = stateFrom;
        this.stateTo = stateTo;
    }

    public String getMessage() {
        return "Invalid state transition from state " +
            stateFrom + " to state " + stateTo;
    }
}

// ---------------------------------------------------------
```

```
// SharedObject.java
// -----------------

import java.util.*;

public class SharedObject{
    private static Vector<SynthesisObject> O =
        new Vector<SynthesisObject>();
    private static Vector<SynthesisObjectPair> I =
        new Vector<SynthesisObjectPair>();

    private static int writeCount = 0;
    private static int readCount = 0;
    private static int readWait = 0;

    private static Semaphore readSem = new Semaphore(0);
    private static Semaphore writeSem = new Semaphore(0);
    private static Semaphore mutex = new Semaphore(1);

    private static Semaphore tokenizer = new Semaphore(1);

    private static boolean checkI(SynthesisObject o1,
            SynthesisObject o2) {
        return true;
    }

    public static void makeI(SynthesisObject o) {
        boolean sleep = false;
        mutex.P();
        if ( (readCount > 0) || (writeCount > 0) )
            sleep = true;
        ++writeCount;
        mutex.V();
        if ( sleep )
            writeSem.P();
        for ( int j = 0; j < O.size(); j++ )
            if ( checkI(o, O.get(j)) )
                I.add(new SynthesisObjectPair(o, O.get(j)));
        O.add(o);
        mutex.P();
        --writeCount;
        if ( writeCount > 0 )
```

```
        writeSem.V();
    else if ( (writeCount == 0) && (readWait > 0) ) {
        --readWait;
        readSem.V();
    }
    mutex.V();
}

public static void removeI(SynthesisObject o) {
    boolean sleep = false;
    mutex.P();
    if ( (readCount > 0) || (writeCount > 0) )
        sleep = true;
    ++writeCount;
    mutex.V();
    if ( sleep )
        writeSem.P();
    SynthesisObjectPair sop;
    for ( int i = I.size() - 1; i >= 0; i-- ) {
        sop = I.get(i);
        if ( sop.isInRelation(o) ) {
            I.remove(sop);
            sop.getRelationObject(o).stopTokenWait();
        }
    }
    O.remove(o);
    mutex.P();
    --writeCount;
    if ( writeCount > 0 )
        writeSem.V();
    else if ( (writeCount == 0) && (readWait > 0) ) {
        --readWait;
        readSem.V();
    }
    mutex.V();
}

public static int getICount() {
    return I.size();
}

public static SynthesisObjectPair getI(int index) {
    boolean sleep = false;
```

```
    mutex.P();
    if ( writeCount == 0 )
        ++readCount;
    else {
        ++readWait;
        sleep = true;
    }
    mutex.V();
    if ( sleep )
        readSem.P();
    SynthesisObjectPair retVal = null;
    try {
        retVal = I.get(index);
    } catch (Exception e) {}
    mutex.P();
    --readCount;
    if ( (readCount == 0) && (writeCount > 0) )
        writeSem.V();
    mutex.V();
    return retVal;
}

public static void captureTokens(SynthesisObject o) {
    boolean localTokenWait;
    while ( !captureTokensInternal(o) ) {
        localTokenWait = o.startTokenWait();
        stopTokenWaitConnected(o);
        if ( localTokenWait )
            o.doTokenWait();
    }
    o.setTokenCollect(false);
}

private static boolean captureTokensInternal(
        SynthesisObject o) {
    o.setTokenCollect(true);
    boolean result = true;
    tokenizer.P();
    try {
        SynthesisObjectPair sop;
        for ( int i = SharedObject.getICount() - 1;
              i >= 0; i-- ) {
            sop = SharedObject.getI(i);
```

```
                if ( sop != null && sop.isInRelation(o) )
                    if ( !sop.captureToken(o) ) {
                        releaseTokens(o);
                        result = false; break;
                    }
            }
        }
        finally {
            tokenizer.V();
        }
        return result;
    }

    public static void releaseTokens(SynthesisObject o) {
        SynthesisObjectPair sop;
        for ( int i = SharedObject.getICount() - 1;
              i >= 0; i-- ) {
            sop = getI(i);
            if ( sop != null && sop.isInRelation(o) )
                sop.releaseToken(o);
        }
        stopTokenWaitConnected(o);
    }

    public static void stopWaitConnected(SynthesisObject o) {
        SynthesisObjectPair sop;
        for ( int i = I.size() - 1; i >= 0; i-- ) {
            sop = getI(i);
            if ( sop != null && sop.isInRelation(o) )
                sop.getRelationObject(o).stopWait();
        }
    }

    public static void stopTokenWaitConnected(SynthesisObject o) {
        SynthesisObjectPair sop;
        for ( int i = I.size() - 1; i >= 0; i-- ) {
            sop = getI(i);
            if ( sop != null && sop.isInRelation(o) )
                sop.getRelationObject(o).stopTokenWait();
        }
    }
}
```

```
// ------------------------------------------------------------

// SynthesisObject.java
// --------------------

public interface SynthesisObject {
    public int getState();
    public void setState(int state)
            throws InvalidStateTransitionException;
    public void stopWait();
    public boolean getTokenCollect();
    public void setTokenCollect(boolean tokenCollect);
    public boolean startTokenWait();
    public void doTokenWait();
    public void stopTokenWait();
}

// ------------------------------------------------------------

// SynthesisObjectPair.java
// ------------------------

public class SynthesisObjectPair {
    private SynthesisObject leftObj;
    private SynthesisObject rightObj;
    private SynthesisObject token;
    private SynthesisObject v1;
    private Semaphore tokenCapturer = new Semaphore(1);

    public SynthesisObjectPair(SynthesisObject leftObj,
        SynthesisObject rightObj) {
        this.leftObj = leftObj;
        this.rightObj = rightObj;
        token = rightObj;
    }

    public SynthesisObject getRelationObject(SynthesisObject o) {
        return o == leftObj ? rightObj : leftObj;
    }

    public boolean isInRelation(SynthesisObject o) {
        return (o == leftObj) || (o == rightObj);
    }
```

```
    public boolean captureToken(SynthesisObject o) {
        tokenCapturer.P();
        if ( token == null && (o == leftObj || o == rightObj) )
            token = o;
        tokenCapturer.V();
        return token == o;
    }

    public boolean releaseToken(SynthesisObject o) {
        tokenCapturer.P();
        if ( token == o )
            token = null;
        tokenCapturer.V();
        return token == null;
    }

    public void setV1(SynthesisObject value) {
        if ( (value != leftObj) && (value != rightObj) )
            throw new RuntimeException(
                "Bad value for SharedObject");
        v1 = value;
    }

    public SynthesisObject getV1() {
        return v1;
    }
}

// ---------------------------------------------------------

// Patient.java
// ------------

import java.util.*;

public class Patient implements SynthesisObject {
    public final int N = 1, W = 2, E = 3;
    private Semaphore blocker = new Semaphore(0);
    private boolean wait = false;
    private Semaphore tokenBlocker = new Semaphore(0);
    private boolean tokenWait = false;
    private boolean tokenCollect = false;
```

```
private Semaphore tokenRereleaser = new Semaphore(1);
private static java.util.Map<Integer, Integer> M =
    new java.util.TreeMap<Integer, Integer>();
private int state;

public Patient() {
    setStateVariable(N);
    SharedObject.makeI(this);
}

public void setTokenCollect(boolean tokenCollect) {
    tokenReleaser.P();
    this.tokenCollect = tokenCollect;
    tokenRereleaser.V();
}

public boolean getTokenCollect() {
    return tokenCollect;
}

public void stopWait() {
    if ( wait ) {
        wait = false;
        blocker.V();
    }
}

public boolean startTokenWait() {
    boolean result = false;
    tokenReleaser.P();
    if ( tokenCollect ) {
        tokenWait = true;
        result = true;
    }
    tokenReleaser.V();
    return result;
}

public void doTokenWait() {
    tokenBlocker.P();
}

public void stopTokenWait() {
```

```
    tokenReleaser.P();
    if ( tokenWait ) {
        tokenWait = false;
        tokenBlocker.V();
    }
    tokenCollect = false;
    tokenReleaser.V();
}

public void destroySynchronization() {
    SharedObject.captureTokens(this);
    SharedObject.removeI(this);
    SharedObject.releaseTokens(this);
}

public int getMapping(int state) {
    Integer ret = M.get(new Integer(state));
    return ret == null ? state : ret.intValue();
}
public void setMapping(int state, int stateMap) {
    M.put(new Integer(state), new Integer(stateMap));
}

public int getState() {
    return state;
}

protected void setStateVariable(int value) {
    state = value;
}

public void setState(int value)
        throws InvalidStateTransitionException {
    boolean succed = false;
    SynthesisObjectPair sop;
    SynthesisObject so;
    int stateRel;
    if ( !(((getState() == N) && (value == W)) ||
           ((getState() == W) && (value == E)) ||
           ((getState() == E) && (value == N))) )
        throw new InvalidStateTransitionException(
    getState(), value);
    while ( !succed ) {
```

```
succed = true;
SharedObject.captureTokens(this);
try {
    if ( (getState() == N) && (value == W) ) {
        for ( int i = SharedObject.getICount() - 1;
              i >= 0; i-- ) {
            sop = SharedObject.getI(i);
            if ( sop.isInRelation(this) ) {
                so = sop.getRelationObject(this);
                stateRel = getMapping(so.getState());
                if ( !(stateRel == W) &&
                     !((stateRel == N) ||
                       (stateRel == E)) )
                    succed = false;
            }
        }
        if ( succed )
            for ( int i = SharedObject.getICount() - 1;
                  i >= 0; i-- ) {
                sop = SharedObject.getI(i);
                if ( sop.isInRelation(this) ) {
                    so = sop.getRelationObject(this);
                    stateRel = getMapping(
                        so.getState());
                    if ( stateRel == W )
                        sop.setV1(so);
                }
            }
    }
    if ( (getState() == W) && (value == E) ) {
        for ( int i = SharedObject.getICount() - 1;
              i >= 0; i-- ) {
            sop = SharedObject.getI(i);
            if ( sop.isInRelation(this) ) {
                so = sop.getRelationObject(this);
                stateRel = getMapping(so.getState());
                if ( !((stateRel == N) ||
                     ((stateRel == W) &&
                     (sop.getV1() == this))) ) {
                    succed = false;
                }
            }
        }
```

```
                }
                if ( (getState() == E) && (value == N) ) ;
            }
            finally {
                if ( !succed )
                    wait = true;
                SharedObject.releaseTokens(this);
            }
            if ( !succed )
                blocker.P();
        }
        setStateVariable(value);
        SharedObject.stopWaitConnected(this);
    }
}

// -------------------------------------------------------------

// Severe.java
// -----------

public class Severe extends Patient {
    public static int S = 4;

    public Severe() {
        super.setMapping(S, N);
    }

    public void setState(int value)
            throws InvalidStateTransitionException {
        if ( getState() == W && value == S ||
             getState() == S && value == N )
            setStateVariable(value);
        else
            super.setState(value);
    }
}

// -------------------------------------------------------------

// Human.java (an example computation code)
// Only the skeleton and the synchronization calls are generated!
// -----------
```

```
public class Human extends Thread {
    private int number;
    private double sickLevel;
    private static TriageNurse tn = new TriageNurse();
    private Patient synch;

    private int tick;

    public static void main(String[] args) {
        Human t;
        for ( int i = 0; i < 3; i++ ) {
            t = new Human(i);
            t.start();
        }
    }

    public Human(int number) {
        this.number = number;
    }

    protected void doSomething() {
        try{
            Thread.sleep((int)(1000 * Math.random()));
        }
        catch (InterruptedException ie) {}
    }

    private void sayFine() {
        System.out.println(number +
            " (tick " + tick + "): I am fine");
    }
    private void saySick(String sickType) {
        System.out.println(number +
            " (tick " + tick + "): I feel " + sickType +
            " bad, should go to the surgery");
    }
    private void sayHeal() {
        System.out.println(number +
        " (tick " + tick + "): OK, the doctor said, I'll heal");
    }
    private void waitForNurse() {
        System.out.println(number +
```

```
        " (tick " + tick + "): in the waiting room");
    try {
        Thread.sleep((int)(1000 * Math.random()));
    }
    catch(InterruptedException ex) {}
}

protected void destroySynchronization() {
    if ( synch != null ) {
        synch.destroySynchronization();
        synch = null;
    }
}

private void iAmSick()
        throws InvalidStateTransitionException {
    destroySynchronization(); // Generated
    synch = new Patient(); // Generated
    saySick("a little bit"); // Not generated
    synch.setState(synch.W); // Generated
    waitForNurse(); // Not generated
    synch.setState(synch.E); // Generated
}

private void iAmSeriouslySick()
        throws InvalidStateTransitionException {
    destroySynchronization(); // Generated
    synch = new Severe(); // Generated
    saySick("seriously"); // Not generated
    synch.setState(synch.W); // Generated
    synch.setState(Severe.S); // Generated
}

private void iFeelBetter()
        throws InvalidStateTransitionException {
    sayHeal();
    synch.setState(synch.N); // Generated
}

public double getSickLevel() {
    return sickLevel;
}
```

```
    public void run() {
        try {
            for ( tick = 0; tick < 10; tick++ ) {
                sayFine();
                doSomething();
                sickLevel = Math.random();
                if ( tn.isSevere(this) )
                  iAmSeriouslySick();
                else
                  iAmSick();
                iFeelBetter();
            }
            destroySynchronization();
        }
        catch (InvalidStateTransitionException iste) {
            System.out.println(iste.getMessage());
        }
    }
}

class TriageNurse {
    public boolean isSevere(Human h) {
        return h.getSickLevel() > 0.75;
    }
}
```

Appendix B

In the following, we show the main building blocks of the graphical tool, which can be used for specifying the synchronization of objects.

Start state. If we want to express that objects are initially in a given state, then we notate it by a dotted style bounded oval box and we write the name of the state in the box. Figure 7.3 describes that every object is initially in its *Normal* state.

If we have more object sets then we label the oval box by the name of the object set between curly brackets, which means that the initial state restriction is only referred to the objects of the given object set. Figure 7.4 shows that sender objects are initially in state *Send* and reader objects in state *Read*.

Common states. A state can be denoted by a solid style bounded oval box as Figure 7.5 shows.

The *true* state. We can refer to any of the states if we write 'true' between quotation marks in a solid style bounded oval box (see Figure 7.6).

Always *true* states. In our modelling language expressions are written. An expression has a source and a destination *synchronization condition*.

Every state is a synchronization condition (it evaluates true if the object is in the state) – these are the primary synchronization conditions. If we have some synchronization conditions then we can connect them with $\vee$, $\wedge$ and $\neg$ operators and we get synchronization conditions.

Figure 7.3: Every object is initially in its normal state.

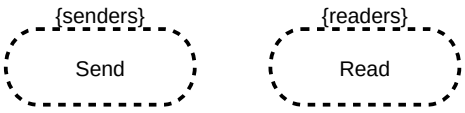

Figure 7.4: Sender objects are initially in state *Send* and reader objects in state *Read*.

If two synchronization conditions are connected with some type of lines (the valid types of lines can be seen below) then we get expressions. Only valid expressions can be written in the synchronization specification. (The parts of the synchronization diagram that do not belong to a valid expression are ignored.)

Because only expressions can be written in the specification, we will show only expressions in the examples. The most simple expression is when a synchronization condition is connected with a *true* state with a solid style line. This means that the synchronization condition is always true during the execution of the program. The most simple synchronization condition is a common state.

If a common state is connected to a *true* state with a solid style line then it means that the given state is always true while the program is running. For example, Figure 7.7 denotes that every object is in normal state during the full lifetime.

Let us remark that implementations can be imagined where an object can be in more states in a time. It depends on the implementation of the functions *getState* and *setState*. If we want objects not to be in more states in a time then we can specify it with this tool (see the example in Appendix A). For the sake of simplicity we generate such a code in which objects may be only in one state at a time. We decided it, because it does not seem to

Figure 7.5: A state named by "State".

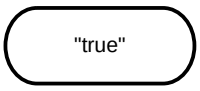

Figure 7.6: Any of the states.

be reasonable in practice that objects can be in more states in one time.

If we want only objects in an object set to be in a state during the execution of the program then we have to define this set in a condition-box (see Figure 7.8) between curly brackets and join this condition to the connector line. In Figure 7.8 we specified that only the objects in set *senders* stay in normal state.

Or-box. If we have two or more synchronization conditions then we can formulate that at least one of them is true. This can be expressed by a dotted style bounded rectangle. If we want to express that at least one of the content of an or-box is true during the execution of the program then we connect it to a true-box with a solid style line. For example Figure 7.9 describes that objects are in state *Normal* or *Send*.

Not-box. If we have a synchronization condition then we can negate it. This can be expressed by a solid style bounded hatched rectangle. If we want to express that the content of a not-box evaluates false during the execution of the program then we connect it to a true-box with a solid style line. For example, Figure 7.10 describes that objects are never in state *Normal*.

And-box. If we have two or more synchronization conditions then we can formulate that all of them are true. It can be done with a solid style bounded rectangle. If we want to express that the content of an and-box is true during

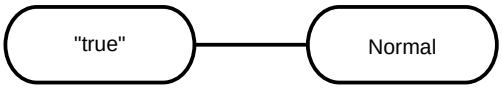

Figure 7.7: Objects are always in normal state.

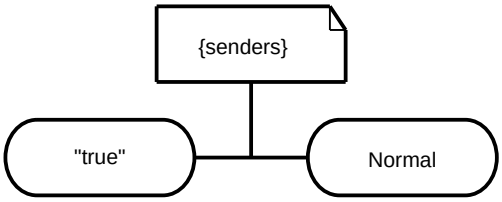

Figure 7.8: Only sender objects are always in normal state.

the execution of the program then we connect it to a true-box by a solid style line. Figure 7.11 describes that objects cannot be in state *Normal* and state *Send* simultaneously.

Equivalence. If we have two synchronization conditions then we can express that they are equivalent. This can be done by connecting the two expressions with a solid style line. Usually we use this for expressing that a condition is equivalent with the negation of an other one. Figure 7.12 expresses that if an object is in state *Send* then it cannot be in state *Normal* but if it is not in state *Normal* then it must be in state *Send*.

Next states. We can make assertions and conditions for the next state of an object. We can express that if the scheduler selects the object for running then it can always enter a well defined destination state from the source state if it decides to enter that destination state (object can decide to enter an other state but a step to an other state cannot be surely executed). Of course, instead of the states we can write an arbitrary synchronization condition. It can be denoted by connecting the two conditions with a solid style line which connects to the source synchronization condition with a special termination, which forks three ways from the source condition as Figure 7.13 shows.

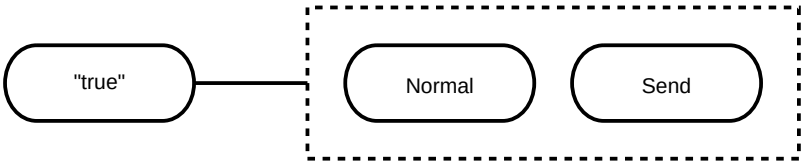

Figure 7.9: Objects are in state *Normal* or *Send*.

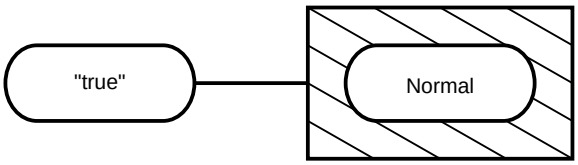

Figure 7.10: Objects never can be in normal state.

Of course, we can make it be referred to only an object set as Figure 7.14 shows.

There are cases when the scheduler chooses an object to run but, the object cannot run because it has to wait for an other object. However, if it can run then it enters a well defined state. It can be expressed with a solid style line which joins the source and the destination conditions and connects to the start synchronization condition with a special termination, which connects with three branches to the source condition (see Figure 7.15).

It can happen that we want to express that if an object is in a state A and the scheduler selects it to run then it can step and the next state has to be state B. It can be done by writing both of the two previous expressions or we can define this more simply by connecting the two synchronization conditions with both of the previously mentioned lines. Figure 7.16 shows an example for this.

Eventualities. We can make conditions not only for next states but for states that objects can reach in the future. We can express that an object reaches a state in the future. It can be expressed by joining the two synchronization conditions with a dotted style line which connects to the source condition with a special termination, which connects with three branches to

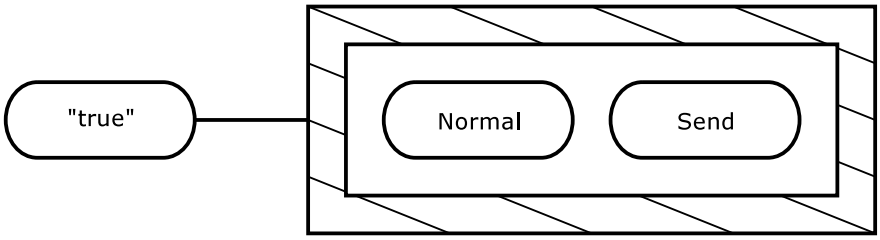

Figure 7.11: Objects cannot be in state *Normal* and state *Send* simultaneously.

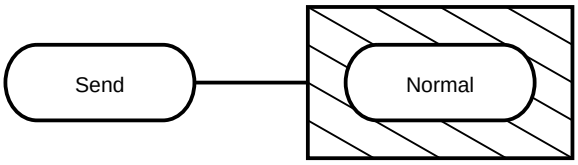

Figure 7.12: If an object is in state *Send* then it cannot be in state *Normal* and if it is not in state *Normal* then it must be in state *Send*.

the source condition. Figure 7.17 shows an example for this.

Figure 7.18 shows an example for the case when we want to express that an object surely reaches state *Normal* from state *Send*, but it can happen that it has to wait for some other objects and it cannot step immediately, but if it can step then the next state will be state *Normal*.

Object connections. If we want to refer to an object and to an object that is connected to the previous one then a solid style bounded oval box has to be drawn, which is labeled with text "other" between square brackets. For example, Figure 7.19 expresses that an object cannot be in state *Send* while at least one other object that is connected to it is in state *Normal*, and reversed similarly, so these two states exclude each other.

We can refer to the steps of another objects with labeling the connector line with label "other" between square brackets. For example, Figure 7.20 expresses that if an object is connected to some other objects and one of the others changes its state then the object stays in its original state.

Figure 7.13: If an object is in state *Normal* and the scheduler chooses it for running then it always can change its state to state *Send*.

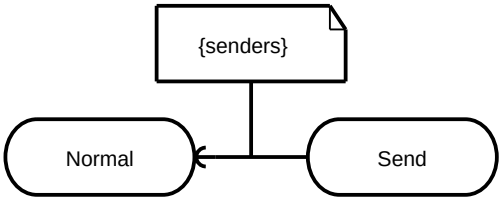

Figure 7.14: If an object that belongs to set *senders* is in state *Normal* and the scheduler selects it for running then it always can change its state to state *Send*.

Blocked states. We can express that an object cannot step to a state if some conditions are true. It can be done by joining the two synchronization conditions with a solid style crossed line which connects to the source condition with a special termination, which forks three ways from the start condition. Figure 7.21 expresses that if an object is in state *Normal* and all of the objects connected to it are not in state *Read*, then the object cannot enter state *Send*.

We can express that an object has to be blocked if some conditions are true. It can be done by joining the source synchronization condition to a true-box with a solid style crossed line with a special termination, which forks three ways from the source condition. Figure 7.22 expresses that if an object is in state *Normal* and all of the objects that are connected to it are not in state *Read* then the object is blocked.

Locally valid conditions. We can bound the validity of expressions with a condition-box (see above). We can write an arbitrary safe first order logic expression into a condition-box and we can connect it to the line that joins the start condition to the destination condition (for the definition of the safe

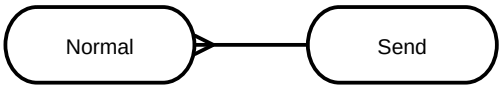

Figure 7.15: If an object is in state *Normal* and it is selected for running then it cannot step or has to step to state *Send*.

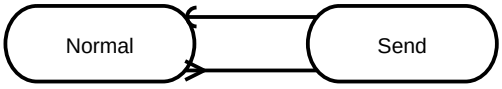

Figure 7.16: If an object is in state *Normal* and the scheduler selects it to run then it can step and the next state has to be state *Send*.

logic formulae see [134]). This means that the expression is valid only if the condition is true for the objects. In the condition-box "this" means the actual object and "other" means a connected object. Furthermore, in a condition-box every function can be used that is defined in the class diagram. Figure 7.23 shows an example when the synchronization constraint is bounded for the objects that have larger index than the original one.

Strong synchronization conditions. If an object checks whether entering an other state is allowed for it and there are no other objects that are connected to it, then the transaction is enabled by definition. But it is possible that there must be at least one object connected to it (for example it sends data to a connected object). We can express that in these cases the object has to wait. It can be expressed by connecting a strong condition-box to the expression. Figure 7.24 shows an example for it.

Of course, it is possible to join several condition-boxes to an expression as Figure 7.25 shows.

Figure 7.17: If an object is in state *Send* then it eventually reaches state *Normal*.

Figure 7.18: If an object is in state *Send* then it eventually reaches state *Normal* and if object can step then it steps immediately to the normal state.

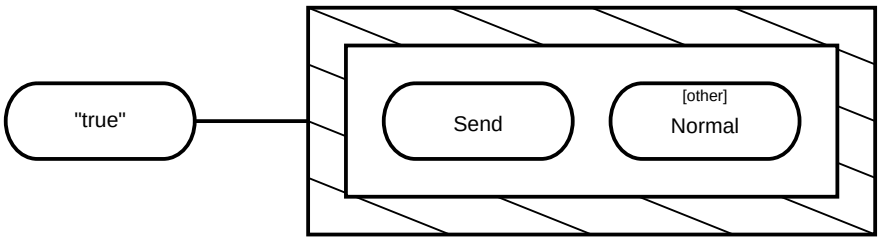

Figure 7.19: An object cannot be in state *Send* while an other object connected to it is in state *Normal* (and it cannot be in state *Normal* while an other is in state *Send*, so these states exclude each other).

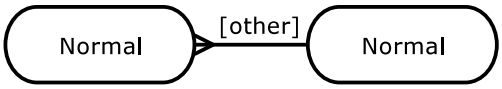

Figure 7.20: If an object is in state *Normal* and an other object that is connected to this changes its state then the object stays in normal state.

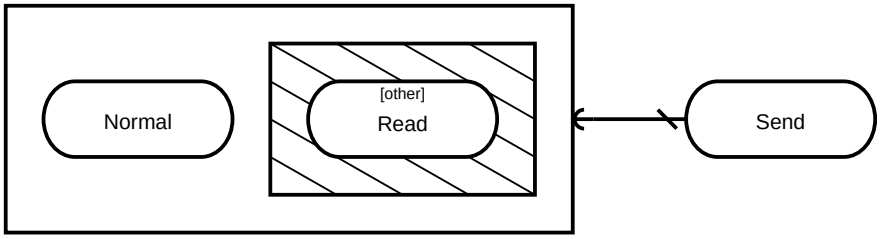

Figure 7.21: If an object is in state *Normal* and all of the objects that are connected to this are not in state *Read* then the object cannot enter state *Send*.

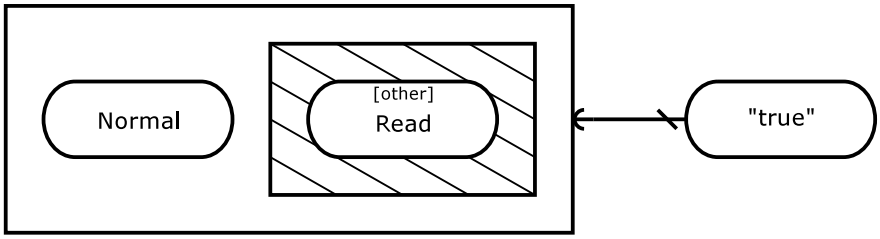

Figure 7.22: If an object is in state *Normal* and all of the objects connected to it are not in state *Read* then the object is blocked.

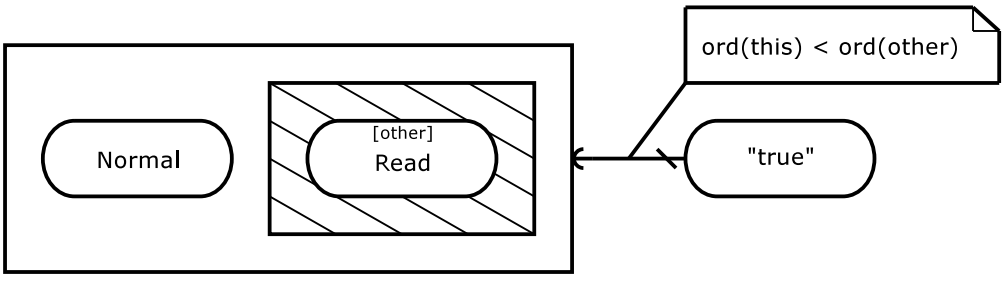

Figure 7.23: If an object is in state *Normal* and all of the objects that are connected to this object and have larger index are not in state *Read* then the object is blocked.

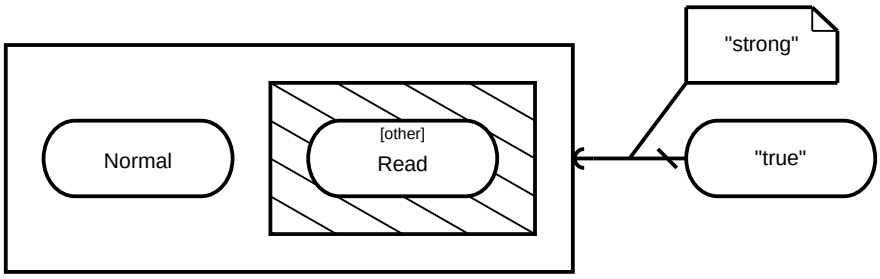

Figure 7.24: If an object is in state *Normal* and all of the objects connected to it are not in state *Read* or there are no connected objects then the object is blocked.

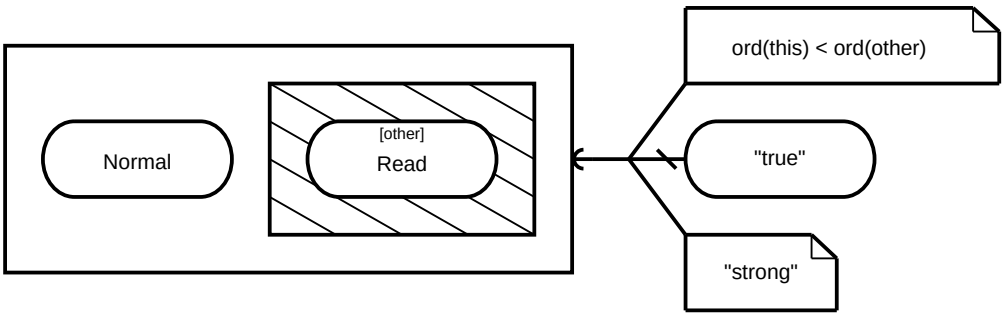

Figure 7.25: If an object is in state *Normal* and all of the objects that are connected to this object and have larger index are not in state *Read* or there are no connected objects then the object is blocked.

Appendix C

Glossary of symbols and acronyms

$\otimes$	Conjunctive guarded command composition operator
$\oplus$	Disjunctive guarded command composition operator
$\bigwedge_i$	Spatial operator which quantifies over every process in O
$\bigwedge_{ij}$	Spatial operator which quantifies over every process pair in I
AOP	Aspect-Oriented Programming
CASE	Computer-Aided Software Engineering
CORBA	Common Object Request Broker Architecture
GI	Global invariant
GIL	Graphical Interval Logics
CTL	Computational Tree Logic
CTL*	An extension for CTL
FNDA	Finite Non Deterministic Automaton
I	Interconnection relation
MPCTL*	Many-Processes CTL*
O	A special vector used to store synchronization objects
OO	Object-oriented
OOP	Object-oriented programming
OOPN	Object-oriented Petri Net
PN	Petri Net
RMI	Remote Method Invocation
RUP	Rational Unified Process
UML	Unified Modelling Language