Annales Univ. Sci. Budapest., Sect. Comp. 22 (2004) 127-150

SYNTHESIS OF A SYSTEM COMPOSED BY MANY SIMILAR OBJECTS

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

Dedicated to Professor Karl-Heinz Indlekofer on his 60th birthday

Abstract. There are different mathematical tools for synthetizing parallel programs. For instance, first order logic, temporal logics and different type algebras are such practical tools. Concurrent systems are special parallel systems in which non-deterministic sequential programs, so called processes are co-operating for the sake of the cause. There are known methods for synthetizing synchronization code for many similar sequential programs running in parallel environment. Some of these methods solve the state explosion problem. We extended a method based on temporal logic to systems consisting of objects. In the first step of the extending the specification of the classes is given by describing the life-cycle of the individual objects. We use temporal logic tools for the sake of the cause. After this objects are made similar to each other in the point of view of synchronization, then code generation possibilities are considered. This method is less powerful than the method for processes, and the code generation is less general, too.

1. Introduction

The increasing popularity of temporal logics over the past decades has promoted the use of different kind of temporal logics for specifying concurrent and distributed systems [2], [3], [6], [7], [8], [14], [16], [17], [22], [23], [24] including object-oriented systems [18], [19], [20], [25], [26],[27], [28], [29], [30],

[31], [32]. During this time different mathematical frameworks for deriving correct concurrent or parallel programs were created [1], [3], [4], [5], [6], [8], [9], [11], [13], [16], [17], [21], [22] [23], [24] as well. Our aim is to present a method for synthetize a system composed by many similar objects from temporal logic specifications.

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

Our method applies the technique suggested by P.C. Attie and E.A. Emerson in [23], and it inherits an important advantage of their method, namely how to deal with an arbitrary number of similar objects without incurring the exponential overhead due to the state explosion problem.

The computation model of Emerson and Clarke's method is based on the concept of concurrent programs of the form $P = P_1||\ldots||P_n$, that consists of a multiple, but finite number of fixed sequential processes active at the same time. Each process is the execution of a sequential program and any two processes are similar if and only if one can be obtained from the other by swapping their indexes. The processes contained within a concurrent program generally have a common goal of meeting the program's specifications. Such processes, therefore, regularly communicate and synchronize activities in order to perform some common operations. However, the code fragments responsible for interprocess synchronization can be generally separated from the sequential application-oriented portions of processes. In this way, the synchronization skeletons can be focused and the details irrelevant to synchronization can be suppressed. The synchronization skeleton of each process P_i is denoted by s_i, the current values of the shared variables $x_1, \ldots, x_m$ are given by a list $v_1, \ldots, v_m$ and the global state of a concurrent program is a tuple of the form $(s_1, \ldots, s_k, v_1, \ldots, v_m)$. A process is given by a finite state machine in the form of a directed graph that consists of nodes labelled by a unique name (s_i), which represents a local state of P_i, and of arcs labelled with synchronization commands of the form $B \rightarrow A$ consisting of an enabling condition B and corresponding action A to be executed.

The computational model based on the object-oriented programming paradigm encourages system (program) builders to consider the artifact under construction as a collection of cooperative objects without shared variables.

To use the method developed by P.C. Attie and E.A. Emerson in [23] we had to solve the handling problem of shared variables by the similar objects. The details can be found in Section 4.3. The rest of this paper

is organized as follows. In Section 2 we look over the main steps of the transformation technique. In Section 3 we shortly look over the tools of temporal logic specification (see the detailed description in [23]). In Section 4 the transformation technique is considered in detail through an example. In Section 5 implementation possibilities are condsidered.

2. Object-oriented synthesis method

Our purpose is to extend Attie and Emerson's synthesis method of process synchronization (the case of processes) [23] to a system of objects (the case of objects). The method produces only the synchronization code, that is sequential application oriented computations are not considered. The synchronization code may be considered as when a process changes the nature of its computations (namely the state of the process changes in terms of synchronization), the process executes its synchronization code which decides whether the process can continue its work.

The technique based on processes defines states, and a process could move from a state to another state. In the case of objects it is more natural: the current values of the properties of the object describe the current state of the object and any change in these properties takes the object to another state. These changes can take place by the methods of the entity, so the synchronization code must be called in every *set* method concerned with the properties. See the details in Section 5.

In the case of processes, where synchronization is applied to pairs of processes and if a process is associated with more than one another process, then the synchronization steps must be executed for every pair in atomic mode. This approach is suitable for objects, too.

In the case of processes similar processes are considered. With every process a unique index i is associated, and the process is referred as P_i. Two processes are considered similar if and only if they differ only in their indexes. It follows from the definition of similarity that the sets of states are identical to two similar processes and there are transitions from and to the same states and the conditions of these transitions are similar. In the case of objects the definition of similarity would excessively restrict the set of synthetizable systems. However, most often this issue can be overcome, so the above definition of similarity can be considered in the case of objects, too. Let us remark that in the case of objects it is considerable to let set I changeable

dynamically because in most cases it is the natural behaviour of systems (for instance, objects are created dynamically).

It is apparent that an object can correspond to a process, so Attie and Emerson's method can be applied to synthetize the synchronization of objects if the correspondence can be done.

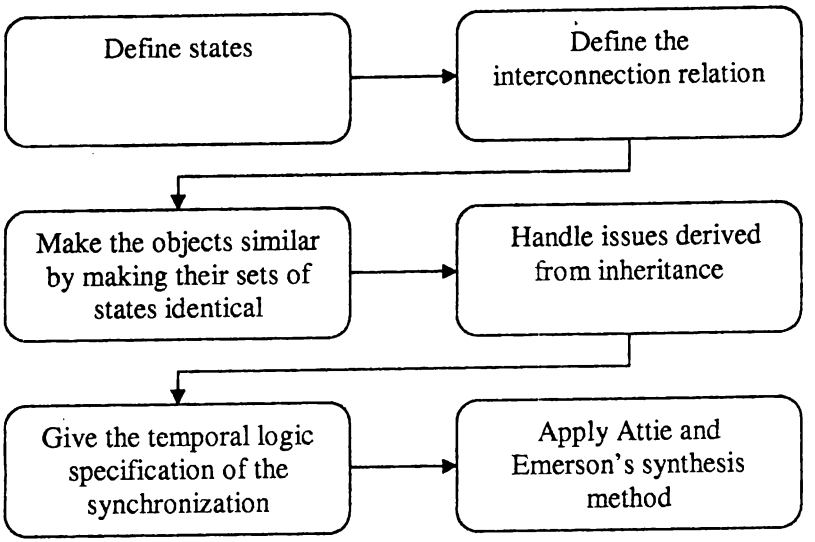

Figure 1. Main steps of the transformation

Let us consider the main steps of the correspondence which will be explained in detail later:

- Considering that an object may have many different states, we distinguish states that are important in terms of synchronization and we introduce an attribute to indicate the current state of the object which is an element of the distinguished states. In this way the number of possible states is reduced reasonably.
- Let us give a method that decides dynamically that which other objects a given object is associated with, namely which other objects it has to be synchronized with, since these associations are not static in a system that consists of objects.
- Let us make the objects similar by making their sets of states identical and giving a global synchronization specification.

- Let us handle issues derived from inheritance by state *set* and *get* methods (see below).

After the above steps individual objects can correspond to processes and the synthesis method of Attie and Emerson can be applied to them.

The steps related to the synthesis of synchronization of systems consisting of objects can be seen in Figure 1.

3. The specification language MPCTL*

In this section we look over the parts in Attie and Emerson's paper [23] that are most important to understand this paper.

The specification language is an extension of the temporal logic CTL* [22], which is a propositional branching-time temporal logic. The basic modalities of CTL* consist of a path quantifier, either A (for all paths) or E (for some path) followed by a linear-time formula, which is built up from atomic propositions, the Boolean operators $\wedge$, $\vee$, $\neg$, and the linear-time modalities G (always), F (sometime), X_j (strong nexttime), Y_j (weak nexttime) and U (until). CTL* formulas are built up from atomic propositions, the Boolean operators $\wedge$, $\vee$, $\neg$, and the basic modalities.

3.1. CTL*

The syntax of CTL* is the following (the state formulas and path formulas can be defined by the following rules):

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

Let us consider the intuitive meaning of the formulas mentioned above. Formula Ef means that there is some maximal path for which f holds; formula Af means that f holds of 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* formulas 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 sequential process i.

A *path* is a sequence of states $(s_1, s_2, \ldots)$ such that $\forall i : (s_i, s_{i+1}) \in R$, and a *fullpath* is a maximal path. A fullpath may be finite or infinite. Let $\pi = (s_1, s_2, \ldots)$ denote a fullpath, then π^i 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 the fullpath π. $M, S \models f$ means $\forall s \in S : M, s \models f$, where S is a set of states. The semantics of $\models$ can be defined inductively:

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 $\pi = (s_1, s_2, \ldots)$ in M such that $M, \pi \models f$;
$M, s_1 \models Af$ iff for every fullpath $\pi = (s_1, s_2, \ldots)$ in M: $M, \pi \models f$;

4. $M, \pi \models f$ iff $M, s_1 \models f$;

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 usual abbrevations for logical disjunction, implication and equivalence can be introduced easily. Furthermore, some additional modalities as abbrevations can be introduced: $Y_j f$ for $\neg X_j \neg f$, Ff for $trueUf$, Gf for $\neg F \neg f$. Y_j is the "process indexed weak nexttime" modality, where the $Y_j f$ formula 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 the Ff formula intuitively means that there is some state along the maximal path where f holds. G is the "always" modality, where the Gf formula intuitively means that f holds at every state along the maximal path.

3.2. The interconnection relation

The interconnection scheme between processes is given by the *interconnection relation I*. $I \subseteq \{i_1, \ldots, i_K\} \times \{i_1, \ldots, i_K\}$, and iIj 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. The term *I-system* is introduced in place of *K-process system*, because there are many possible interconnection schemes for a given number K of processes.

3.3. MPCTL*

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 considering q as an abbrevation 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 $\forall i \in I : M, s \models f_{ij}$.

4. A synthesis method for many similar objects

Analyzing the concurrent programs it can be observed that the parts responsible for interprocess synchronization can be separated from the sequential applications-oriented computations performed by the process. This suggests that we focus our attention on synchronization skeletons which are abstractions of concurrent programs, where details that are irrelevant to synchronization are suppressed. The synchronization skeleton of a process P_i may be viewed as a state-machine, where each state represents a region of sequential computation code and where each arc represents a conditional transition between different

regions. The conditional transitions are used to enforce synchronization constraints.

The synchronization skeleton of each process P_i is formally a directed graph. In the graph each node is labelled by a unique name (s_i) which represents a *local state* of P_i and each arc between two states is labelled by a synchronization command $B \to C$ consisting of an enabling condition B and the corresponding action C to be performed (i.e. a guarded command [10]).

A unique index i is associated with every process P_i. Two processes are *similar* iff one can be obtained from the other by swapping their indexes. Intuitively, this corresponds to concurrent algorithms, where a generic indexed part of code gives the code body for all processes.

A set of states is associated with every object. Objects can change their state among the elements of the associated set during their life-cycle. The specification of the synchronization code is given by temporal logic formulas that make restrictions on the relations of the states. MPCTL* is going to be used as tool for specification, thus all objects should have the same state-set, so we can use spatial operators. Furthermore, we would like to use a synchronization skeleton obtained by the synchronization restrictions of any two objects to produce the synchronization code of the whole system. If the object P_i is in state A, then atomic proposition A_i is true (the associated individual object is indicated by the index of the state). We remark that the I interconnection relation is used to define the pairs of objects that will be synchronized. If the object P_i is associated with a number k of objects which P_i will be synchronized with, then the set I contains exactly k objects in pair with P_i. For example, if the object P_i has to be synchronized with objects $P_{i_1} \dots P_{i_n}$ then $\{(i, i_1), \dots, (i, i_n)\} \subseteq I$.

4.1. One-class system

In the case the system consists of entities (instances) that are objects of the same class, it is obvious that the state-sets of the entities are identical. However, there are many possible interconnection schemes. For instance, suppose that we have a class that has a size property, and we want only entities larger than 2m to be associated. It is clear that it matters which index-pairs get into I. Furthermore, objects can be created dynamically, or the size property of entities can be changed freely, so objects have to take care of getting in set I (with their pairs), and getting out of it. So it is neccessary that objects can access the states of each other in mutual exclusive mode, and only one entity can access the set I simultaneously.

Unfortunately there are more difficulties, since before an object-pair gets into set I its consistency must be checked, namely if either entity of the pair is

in some state that is inconsistent with the state of some object associated with it. In this case the elements of the pair have to wait and can not continue their work. For this reason the possibility of deadlock is increasing; and checking deadlock freedom becomes complicated which is unambigiously associated with the dynamic nature of objects. In the first approach we are not considering deadlocks.

We need a procedure that controls when an object can get into set I. Typically, this procedure (*makeI*) is called with an object (as a parameter) once, when the object is created, and in that moment the object is in some initial state that is permitted in any case, so the object can get into I immediately.

We can describe the pairs in I the most simple way by keeping reference-pairs which point to entity-pairs that will be synchronized.

In order that objects can find out whom they have to be synchronized with, they have to know all participants of the system. So we have to keep a record of references of all the created objects (see *static Vector I* in Figure 2).

It follows from the foregoing that we can introduce a new object that stores the references to all objects, the relation I, and a procedure that controls when an object can get into set I. To make this decision the procedure needs to know the callable methods of the objects, so all the objects have to implement some common interface (or inherit from a common ancestor).

4.2. 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. We can solve this problem by simply taking the union of the state-sets of all classes, and then consider the synchronization code above this set of states. On this way the entities can refer to states of other entities which will never be taken, but are important in terms of synchronization.

Considering that the state unioning 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.

4.3. Using shared variables

The synthesis method of Attie and Emerson [23] generates shared variables in the synchronization code. Handling of these shared variables changes in the case of a system of objects. During the synthesis method number m binary variables are associated with every object-pair which variables can be modified and read by only the objects of the pair. Our system consists of objects, so we can introduce a new class named *SharedObject* which holds the shared variables generated by the method (see $v_1, \ldots, v_m$ in Figure 2). When a new object pair

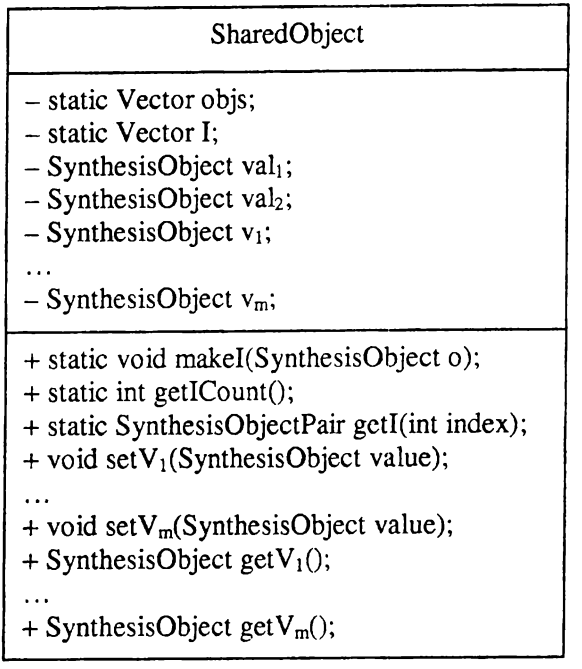

Figure 2. Class SharedObject

gets into I, an instance of SharedObject can be associated with it immediately. The variables in these instances are binary and take the reference of either of the associated object pair. We give methods for setting and getting the values of the variables. Get methods can run parallelly, but no set method can run simultaneously with another set or get method. The possible two values are

known when an instance is created, so we can store these values in the object instance, and the self reference and a true or false value can be passed to the set method depending on which object we want to set the shared variable to (see val_1, val_2 in Figure 2). However, objects refer to states of their associated objects in the synchronization code, so it is practical to store the references of associated pairs at the entities. This way storing the possible values of variables serves only error checking purposes.

Let us use the class *SharedObject* to define I and *objs* lists as class level variables, where I is the interconnection relation and *objs* is a list of the objects in the system. Let us define the procedure *makeI* which monitors the changes of objects and dynamically puts in or removes pairs from I. Suppose that every class inherits from a class named *SynthesisObject* in order that objects could be passed to procedure *makeI* as parameters. Furthermore, suppose that the constructor of *SynthesisObject* calls procedure *makeI*. The class *SharedObject* is shown in Figure 2.

4.4. Specification

Specifying a system composed by 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. Perform the combining of the state-sets of the classes. We can assume that the sets of states of any pair from the individual classes are disjunct. Thus combining the state-sets is actually producing the disjunct union of the individual state-sets.
2. Reduce the sets by merging the states of different types which formulate identical constraints for the object.
3. Let us give the set I with the procedure that decides that which other objects an object is associated with. The procedure may be a class function of some given class and this procedure must be called in mutual exclusive mode. Let this procedure be a static method of the class introduced to handle the shared variables. Procedures that set or get the values of the variables can not run simultaneously with the previous procedure. Assume that all types are subtypes of a predefined type which has the attributes needed to determine the set of associated pairs, and assume that its constructor calls the previous procedure.
4. Give the temporal logic specification in MPCTL*.

4.5. Example

A group of experts is examining lions, so they capture and keep lions in cages, and after the analysis the lions are set free. The group has only one cage, so they have to be aware that no two lions may be in the cage simultaneously, so they can not hurt each other. Therefore, the state of an animal can change from *free* to *captured*, then change to *caged* while it is examined, and then its state changes to free again. See the class diagram of the example in Figure 3.

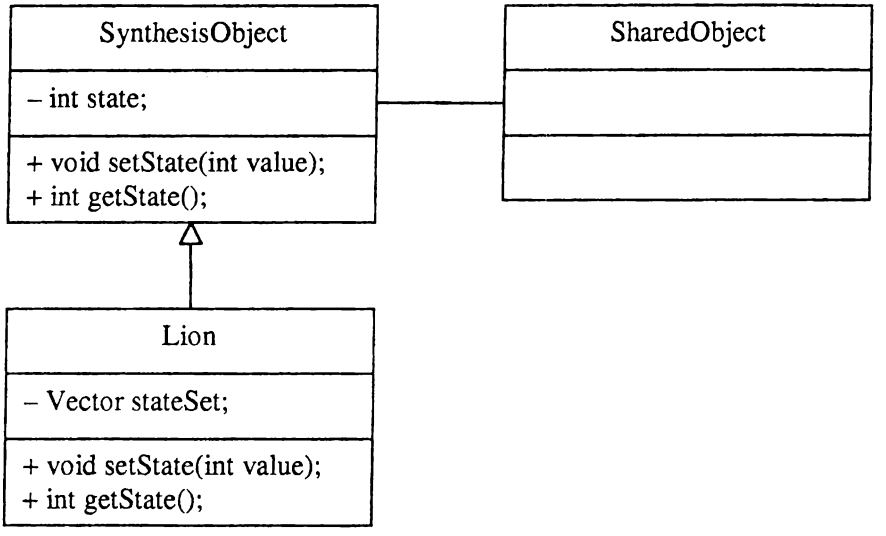

Figure 3. Class diagram of the example

The system consists of instances of only one class, so we do not have to bother with combining the state-sets. Each entity is connected to all the other entities, so, for example, if we have n items then for item e_n there are $n-1$ pairs named $(e_n, e_1) \ldots (e_n, e_{n-1})$ in the set I. These pairs get into I immediately when the item e_n is created. So the class *lion* inherits from a class that contains a constructor which calls the procedure that expanses I. Hence the procedure may be similar to the following:

```
synchronised public static void makel(SynthesisObject o)
{
    for (int j=0; j < objs.size(); j++ )
```

```
            I.add(new SynthesisObjectPair(o, objs.get(j)));
        objs.add(o);
    }
```

The temporal logic specification: it is clear from the description of the problem that object P_i may be in states N_i, C_i and H_i, where N is normal state, C is captured state and H is analyzed state when the entity is locked in a cage. Restrictions are given by the following formulas:

1. initial state (every process is initially in its normal state). The entities which will enter the system later are not contained in I yet, so they do not cause any problem:
$$\bigwedge_i N_i$$
2. it is always the case that any move that P_i makes from its normal state leads into its captured state, and such a move is always possible:
$$\bigwedge_i AG(N_i \Rightarrow (AY_i C_i \wedge EX_i C_i))$$
3. it is always the case that any move that P_i makes from its captured state leads into its caged state:
$$\bigwedge_i AG(C_i \Rightarrow (AY_i H_i))$$
4. it is always the case that any move that P_i makes from its caged state leads into its normal state, and such a move is always possible:
$$\bigwedge_i AG(H_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$
5. P_i is always in exactly one of the states N_i, C_i or H_i:
$$\bigwedge_i AG(N_i \equiv \neg(C_i \vee H_i))$$
$$\bigwedge_i AG(C_i \equiv \neg(N_i \vee H_i))$$
$$\bigwedge_i AG(H_i \equiv \neg(N_i \vee C_i))$$
6. P_i does not starve:
$$\bigwedge_i AG(C_i \Rightarrow AF H_i)$$

7. a transition by one process cannot cause a transition by another:

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

$$\bigwedge_{ij} AG((C_i \Rightarrow AY_j C_i) \wedge (C_j \Rightarrow AY_i C_j))$$

$$\bigwedge_{ij} AG((H_i \Rightarrow AY_j H_i) \wedge (H_j \Rightarrow AY_i H_j))$$

8. no two processes access caged state together:

$$\bigwedge_{ij} AG(\neg(H_i \wedge H_j))$$

The synchronization skeleton generated by the method of Emerson and Attie is shown in Figure 4, where sign $\oplus$ is an operator on guarded commands and roughly means that one of the actions of the commands can be executed if the associated guard condition evaluates true, otherwise the process blocks. Furthermore, $\otimes$ is as operator on guarded commands that roughly means that if all guard conditions evaluate true, then all commands can be executed parallelly otherwise the process blocks. Details can be found in [23].

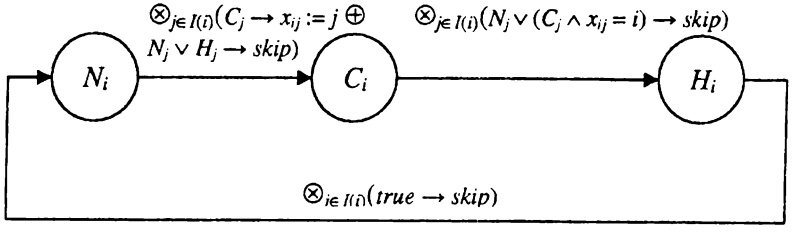

Figure 4. Synchronization skeleton of the example

4.6. Handling inheritance

If a class is a descendant of another 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 than in the ascendant and we might keep the advantages of polymorphism. Suppose, for example, that we have class Animal, which has states free, captured and caged, and we have classes lion and zebra which inherit from animal. We would like to describe that no lion can be in one cage

with a zebra simultaneously, but any number of zebras can stay together. It is clear that the states have to be handled different ways in the case the object is a lion or a zebra, so we cannot define the specification.

There is a solution for this problem if we consider states as value returned by some function, and we override this function in the descendants. In this case we read the proper value in both cases according to the dynamic type of the object. Similarly, if a procedure sets the state of the objects, that procedure can be overridden in the descendants, too. Considering the previous example we can override the state procedure in such a way that it returns "caged-lion" state if the object is a lion and it is in caged state. This way we can describe the restrictions that if a lion is in a cage then no zebra associated with the lion can be in the same cage simultaneously.

Suppose that the previously introduced *SynthesisObject* class has a *state* attribute and *getState* and *setState* methods. The *state* attribute describes the actual state of the object, while the associated set and get methods serve the purpose of reading and writing its value. We use integer values to represent the states in the source code.

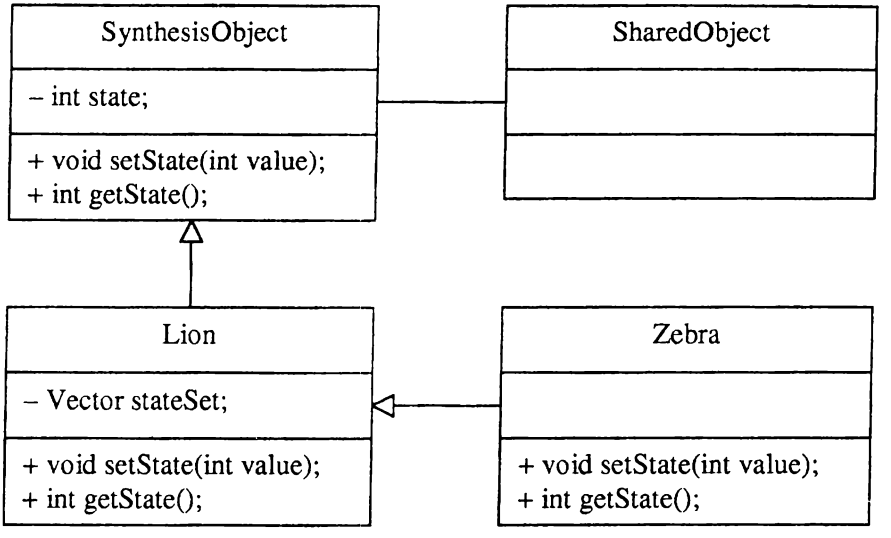

Figure 5. Class diagram of the example

4.7. Example of handling inheritance

Let us suppose that the scientist group in the previous example would like to examine zebras in the future. They know that zebras do not hurt each other, so they can keep zebras in one cage. However no animal can be in one cage with a lion. See the class diagram of the example in Figure 5.

In this case the following states can be associated with an entity:

- In the case of a lion: normal (N), captured (C), caged (H).
- In the case of a zebra: normal (N), captured (C), caged (shared) (Z).

Uniting the sets of states means that the states of entity P_i are in set $\{N, C, H, Z\}$ (the appropriate atomic propositions are N_i, C_i, H_i, Z_i).

The body of method *makeI* is unchanged since each entity is associated with each other in the future, too, and the method is called by every object when the object is created.

The temporal logic specification of the system is the following:

1. initial state (Each process is initially in its normal state. The statement is trivially true for the objects that are not entered the system yet, because they are not in I):
$$\bigwedge_i N_i$$
2. it is always the case that any move that P_i makes from its normal state leads into its captured state, and such a move is always possible:
$$\bigwedge_i AG(N_i \Rightarrow (AY_i C_i \wedge EX_i C_i))$$
3. it is always the case that any move that P_i makes from its captured state leads into its caged or shared caged state:
$$\bigwedge_i AG(C_i \Rightarrow (AY_i(H_i \vee Z_i)))$$
4. it is always the case that any move that P_i makes from its caged state leads into its normal state, and such a move is always possible:
$$\bigwedge_i AG(H_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$
5. it is always the case that any move that P_i makes from its shared caged state leads into its normal state, and such a move is always possible:
$$\bigwedge_i AG(Z_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

6. P_i is always in exactly one of the states N_i, C_i, H_i or Z_i:

$$\bigwedge_i AG(N_i \equiv \neg(C_i \vee H_i \vee Z_i))$$
$$\bigwedge_i AG(C_i \equiv \neg(N_i \vee H_i \vee Z_i))$$
$$\bigwedge_i AG(H_i \equiv \neg(N_i \vee C_i \vee Z_i))$$
$$\bigwedge_i AG(Z_i \equiv \neg(N_i \vee C_i \vee H_i))$$

7. P_i does not starve:

$$\bigwedge_i AG(C_i \Rightarrow AF(H_i \vee Z_i))$$

8. a transition by one process cannot cause a transition by another:

$$\bigwedge_{ij} AG((N_i \Rightarrow AY_j N_i) \wedge (N_j \Rightarrow AY_i N_j))$$
$$\bigwedge_{ij} AG((C_i \Rightarrow AY_j C_i) \wedge (C_j \Rightarrow AY_i C_j))$$
$$\bigwedge_{ij} AG((H_i \Rightarrow AY_j H_i) \wedge (H_j \Rightarrow AY_i H_j))$$
$$\bigwedge_{ij} AG((Z_i \Rightarrow AY_j Z_i) \wedge (Z_j \Rightarrow AY_i Z_j))$$

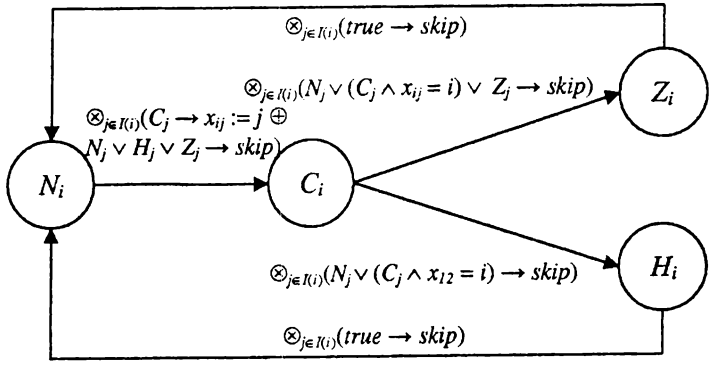

Figure 6. Synchronization skeleton of the example

9. no two processes access caged state together:

$$\bigwedge_{ij} AG((\neg(H_i \wedge H_j)) \wedge (\neg(H_i \wedge Z_j)) \wedge (\neg(Z_i \wedge H_j)))$$

The synchronization skeleton generated by the method related to systems consists of objects is shown in Figure 6. The notation X_i means that object i is in state X, namely $((SynthesisObject)objs.get(i)).getState() == X$.

A lion never can be in state Z and a zebra never can be in state H, so these states may be removed from the synchronization skeletons of the appropriate objects. The result is shown in Figure 7.

5. Implementation

In the implementation phase our task is to produce method *setState* of class *SynthesisObject* which checks whether the state transition is possible in the actual circumstances.

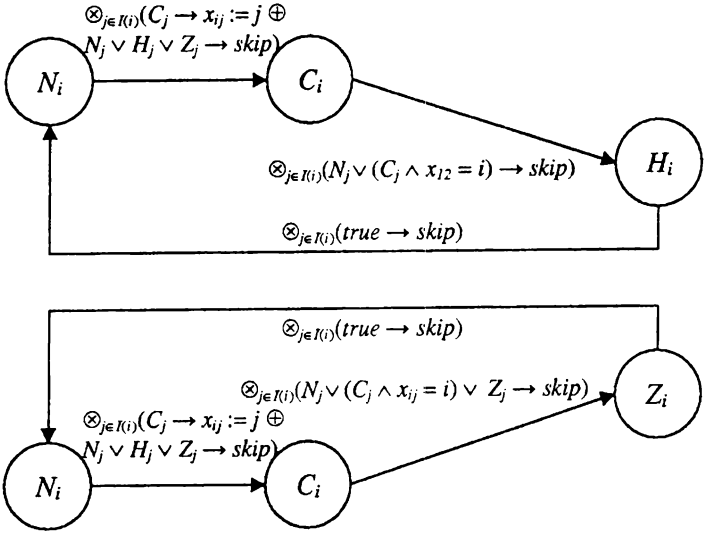

Figure 7. Final synchronization skeleton of Lion (above) and Zebra (below)

Let us consider a simpler issue before we start dealing with mutual exclusion of the variables, namely the problem of writing and reading I. The

methods used for reading and writing I can be given, too; they are practically static methods of class *SharedObject*.

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

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

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

Let us deal with the evaluation of conditions, namely the method *setState* in the following. To produce method *setState*, the abstract program of the

synchronization, which is a finite deterministic automata, is given by the algorithm. Then we make the condition checking part on the basis of the conditions in the automata and if a given condition is fulfilled then we execute the action part associated with the condition. The automata may be given by a list of the transitions. Only one transition can be generated by the synthesis between two states, so a transition may be built from the following elements:

- start state,
- end state,
- condition (in Polish form expression in order to simplify the evaluation),
- the list of the operations on the shared variables.

We have to solve the problem of synchronization of the condition evaluation and the execution of the actions belonging to the conditions. The method *setState* uses the values of the shared variables and may change the variables, too, in case the transition is enabled. That is why the shared variables should be changed by at most one object simultaneously. Let us notice that this restrictioin 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 can not start evaluating a condition while another object is trying to process a transition (namely, the object has started the evaluation and has not done the action).

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

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

Possibility of deadlock arises in progress of obtaining tokens. Deadlock can be avoided if an object drops all tokens that it owns if it tried to get a token from an object that is already waiting for a token, and the object restarts obtaining token some time later after dropping. It is clear that this implementation may lead to livelock: let us suppose that the objects a, b and c are going to obtain tokens from each other. Let a get the token from b, b

from c and c from a. Then let a ask the token from c. It is not possible, so a drops all the tokens it owns. Then let c try to get the token from b. It fails so c drops its tokens, too. Then only b has any token. Then let a get token from b and c from a, then start this process again with simple modification so that c will be the only object that owns any tokens. And so on.

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

In the second method the objects keep a record of the number of tokens got. In case an object obtains the token from another object, that is collecting tokens, too, then the result of the transaction depends on the number of tokens the objects already got. The object that owns the less number of tokens drops them and after some time elapsed restarts collecting tokens. It is clear that this method is appropriate for avoiding livelock, too.

6. Future work

It is found out that the synthesis method of Attie and Emerson [23] can be extended to systems of objects, but the increasing number of states has to be taken into account. This effect may complicate the system and it leads to extra work because we have to try to contract different states.

Considering that the synchronization skeleton of individual objects may contain states which can never be taken, the deadlock checker algorithm (the algorithm is detailed in [23]) may result that deadlock is possible, nevertheless deadlock freedom would be set out in the original system. Consequently, deadlock checking possibilities and extra work needed to manage the above issue should be considered.

As above mentioned, some parts of the resulted synchronization skeleton may be removed. This gives the idea to describe the system by local specifications of the classes and synthetize the synchronization from the local specifications instead of the global specification of the whole system. The possibilities of this alternative specification may be considered.

The correctness and completeness of the method should be considered, too. This research is based on methods of formal semantics. Semantics can be associated with the individual methods, and correctness and completeness can be proved based on these definitions.

The possibilities of embedding temporal logic specifications into different object-oriented designer softwares may be considered, too.

References

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

[2] **Chaochen Z.,** Specifying communicating systems with temporal logic, *LNCS* **389** (1987), 304-323.

[3] **Horváth Z.,** The weakest precondition and the specification of parallel programs, *Proc. of the Third Symposium on Programming Languages and Software Tools, Kaariku, Estonia, 1993,* 24-34.

[4] **van Lamsweerde A. and Sintzoff M.,** Formal derivation of strongly correct concurrent programs, *Acta Informatica,* **12** (1) (1979), 1-31.

[5] **Lisper B.,** Synthesizing synchronous system by static scheduling in space-time, *LNCS* **362** (1987).

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

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

[8] **Wolper P.,** The tableau method for temporal logic: An overview, *Logique et Anal.,* **28** (1985), 119-136.

[9] **Gopalakrishnan G. and Fujimoto R.,** Design and verification of the rollback, Chip using HOP: A case study of formal methods applied to hardware design, *ACM Trans. on Comp. Syst.,* **11** (2) (1993), 109-145.

[10] **Dijkstra E.W.,** *A discipline of programming,* Prentice-Hall, Englewood Cliffs, 1976.

[11] **Owicki S. and Gries D.,** An axiomatic proof technique for parallel programs, *Acta Informatica,* **6** (1976), 319-340.

[12] **Hoare C.A.R.,** Communicating sequential processes, *Comm. ACM,* **21** (1978), 666-677.

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

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

[15] **Smullyan R.M.,** *First order logic,* Springer, 1971.

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

[17] **Chandy K.M. and Misra J.,** *Parallel program design: A foundation,* Addison-Wesley, 1989.

[18] **Misra J.,** Specifying concurrent objects as communicating processes, *Science of Computer Programming,* **14** (1990), 159-184.

[19] **Kozma L.,** Shared data abstractions, *Proc. of Fourth Hungarian Computer Science Conference 1985,* eds. M.Arató, I.Kátai and L.Varga, 201-210.

[20] **Kozma L.,** A temporal logic approach to shared data abstractions, *Conf. on Automata, Languages and Progr. Systems, Salgótarján, 1986,* 160-172.

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

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

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

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

[25] **Wegner P.,** Classification in object-oriented systems, *SIGPLAN Notices,* **21** (10) (1986), 173-182.

[26] **Meyer B.,** *Object-oriented software construction,* Prentice Hall Inc., 1988.

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

[28] **Love T.,** *Object lessons,* SIGS BOOKS Inc., New York, 1993.

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

[30] **Booch G.,** *Object-oriented analysis and design with applications,* The Benjamin/Cummings Publishing Company Inc., 1994.

[31] **Holland I.M. and Lieberherr K.J.,** Object-oriented design, *ACM Computing Surveys,* **28** (1) (1996), 273-275.

[32] **Wieringa R.,** A survey of structured and object-oriented software specification methods and techniques, *ACM Computing Surveys,* **30** (4) (1998), 459-527.

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