6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Synthesis of the synchronization of pipeline systems

Balázs Ugron[a], Szabolcs Hajdara[b]

[a]Department of Software Technology and Methodology,
Eötvös Loránd University
e-mail: balee@inf.elte.hu

[b]Department of Software Technology and Methodology,
Eötvös Loránd University
e-mail: sleet@inf.elte.hu

Abstract

The pipeline systems and different subtypes of pipelines are interesting parts of parallel systems in software engineering. That is why it seems to be worth to deal with the possibilities of the specification of the synchronization of these systems.

Different methods exist that can be used to synthesize the synchronization of parallel systems based on some kind of specification, but these methods can not be applied directly for pipeline systems, because of some special properties of the pipeline systems, and the methods themselves.

The method that seems to be the most promising is the method of Attie and Emerson, which is a synthetization method for many similar processes based on a special temporal logic specification.

In this paper we give an extension for this method, so that the extended method is able to handle more properties of parallel systems, especially of pipeline systems. Furthermore, we give an abstract synchronization of the linear pipeline system.

Categories and Subject Descriptors: D.2.1 [Software engineering]: Requirements specification; F.3.1 [Logic and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs – *assertions, invariants*; F.4.1 [Mathematical Logic]: Temporal logic.

Key Words and Phrases: semantic tableaux, pipeline, synthesis, parallel systems, temporal logic.

1. Introduction

In the following, we will consider the synchronization possibilities of a special part of parallel systems, the pipeline system. As usual (see [1], [2]), we consider only the synchronization part of the processes, because the real computation code usually can be separated from the synchronization part of parallel systems.

A pipeline system is a parallel system of processes, which is built in order to solve some problem. The processes in the system are aligned to a row by the connections between them, so every process in the system has exactly two connections, except the first and the last processes, which have only one. The processes between the two ends work on similar tasks, so their synchronization are obviously similar, too.

There are methods in the literature, which can be used to synthesize the synchronization part of a system from temporal logic specification, but these methods can not be directly applied in this case. For example, the method of Emerson and Clark [1] suffers from the so called state explosion problem [2], so it can not be applied for a large number of processes, in pratice. Another example is the method of Attie and Emerson [2] which can handle large systems, but this method can be only used for systems consisting many similar processes, and this is not that case.

In this article, after a short description of the synchronization of many similar processes [2], we will introduce an extension of the method, with which it will be possible to handle the case of pipeline systems, too.

2. Synthesis of many similar processes

In this section we look over the parts in Attie and Emerson's paper [2] that are most important to understand this paper. The reader will generally find only informal definitions in this section, the exact definitions can be found in [2].

At first, Attie and Emerson's method specifies, that the processes must be similar. In this case, similarity means that any two processes can be exchanged with each other, except their indexes. This restriction is used many times in the method.

2.1. CTL*

The specification language is an extension of the temporal logic CTL* 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.

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 at least the previous state.

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 $\neg F \neg f$.

The reader can find the formal definition of the semantics of CTL* formulas in [2].

2.2. The interconnection relation

The interconnection scheme between processes is given by the *interconnection relation I.* $I \subseteq \{i_1, \dots, i_K\} \times \{i_1, \dots, i_K\}$, and $i\ I\ j$ iff processes i and j are interconnected. I is symmetric and irreflexive relation.

Those process pairs, that are in the interconnection relation, will be synchronized with each other, and the others will not. This means, that the behaviour of the system can be simply changed by the interconnection relation. For example, the synchronization of the eating philosophers problem is the same as for the standard mutual exclusion problem – except the interconneciton relation.

2.3. MPCTL*

An MPCTL* (Many-Process CTL*) formula consists of a spatial modality followed by a CTL* state formula. A spatial modality is of the form $\bigwedge_i$ or $\bigwedge_{ij}$. $\bigwedge_i$ quantifies the process index i which ranges over $\{i_1, \dots, i_K\}$. $\bigwedge_{ij}$ quantifies the process indexes i, j which range over the elements of I.

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 abbrevation and expanding it like

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

2.4. The method

In this section, we give an extremely short informal description of the synthesis method of Attie and Emerson, which will be informative enough to catch the point, though.

At first, the behaviour of the system needs to be specified in the above described temporal logic language, MPCTL*. This specification is applied to an arbitrary process or process pair from the system.

Any known method (for example the one described in [1]) can be used to synthetize the synchronization skeleton of an arbitrary process pair from the system.

Then the method takes advantage of the fact that the processes in the system are similar, and produces a global synchronization skeleton for the whole system, based on the skeleton synthesized for the pair system.

3. Synthesis of a pipeline system

Our main goal in this paper is to develop a method, with which the synchronization skeleton of a pipeline system can be synthetized. The method of Attie and Emerson which we roughly described above, can not be applied direcly in the case of a pipeline system.

The first reason is, that the processes in a pipeline system are not similar. Although the processes except the first and the last one of the pipeline are similar, the mentioned two processes differ from them.

The second reason is, that the communication in a pipeline system has a direction – from the start of the pipeline to the end. This method does not provide facilities to distinguish the processes even in the specification, so we can not handle directions, so the synthetized synchronization code will not be efficient.

At first, we give an extension for the method, with which the side processes can be handled, too. We introduce one more abstraction level in the method: we separate the processes inside the pipeline from the two processes at the ends, and handle them as an embedded system, and synthetize the synchronization for them. After that, we handle the embedded system as a part of new system, besides the two processes at the ends of the pipeline.

3.1. The embedded system

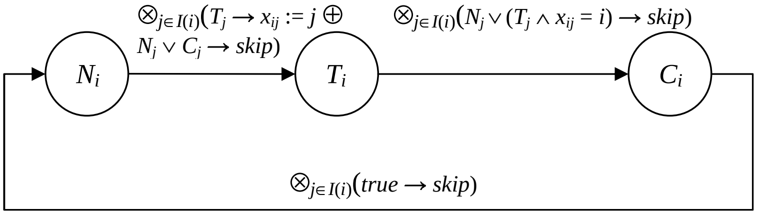
At first we give a straightforward solution for the synchronization of the previously mentioned embedded system in the pipeline. This is a very inefficient approach, but in this case, the method of Attie and Emerson can be applied directly. In fact, this is the solution of the standard mutual exclusion problem.

In this case, the processes will have three states, a normal (N), a trying (T) and a critical (C) state. A process steps in its trying state, when it wants to go to the critical state, and two interconnected processes cannot be in their critical state at the same time. The processes do the communication (data receiving and passing) and the real computation, too, in the critical state.

The synchronization skeleton for such a system is deduced in [2]. The resulted automata can be seen in figure 1.

As we said, this approach is extremely inefficient, because the neighbors of a process can not do anything, while the process is in critical state, although theoretically they would be able to do the computational part of their work simultaneously.

A much better approach will be shown later, in section 3.2.



$\otimes_{j\in I(i)}(T_j \rightarrow x_{ij} := j \oplus N_j \vee C_j \rightarrow skip)$
$\otimes_{j\in I(i)}(N_j \vee (T_j \wedge x_{ij} = i) \rightarrow skip)$
N_i
T_i
C_i
$\otimes_{j\in I(i)}(true \rightarrow skip)$

Figure 1: Synchronization skeleton of the embedded system

3.2. Another approach for the embedded system

We gave an inefficient solution for the embedded system in section 3.1. We introduce a better approach in this section.

As we found out, the main problem in the synthetization of this part of the system is that the method does not allow us to make a distinction between processes, and so we can not express the direction of the communication, so the result will be inefficient.

To get over this issue, we introduce a new definition for the spatial operators defined by Attie and Emerson – or we can say, we define two new spatial operators.

The original definition of the spatial operators can be found in section 2.3. We add a ρ predicate parameter to the spatial operators:

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

This definition intuitively means, that a connection between two processes, that is defined in the interconnection relation may be actual or non actual in different situations, and the actuality of the interconnection is driven by the predicate ρ.

For the sake of effectiveness, there are two critical sections for every process in this approach, a critical section for reading the data from the previous process, and another one for sending the data to the next process. Moreover, there will be a *sent* and a *received* state for each process, because the communication works through shared variables, and the flow of the communication should be driven by the synchonization.

So, the processes will have many states: N (normal), T (try to read), R (read), C (check), W (work), E (try to send), S (send) and finally A (after send). The two critical states are R and S, and the restriction is, that if a process is in its state S, then the following process must not be in its state R, and vice versa.

Let us see, what happens in the states. State N is the start state of every process. State T is a trying state before the R critical section, that is for reading. State C is a checkpoint after the reading. State W is the state in which the process does its real computation work. State E is a trying state before the S

critical section, that is for sending. Finally state A is another checkpoint, now after sending.

Let us see some formulas from the temporal logic specification of the synchronization of the embedded system, in which the parametrized spatial operators are used.

- $\bigwedge_i AG(C_i \Rightarrow (AY_i Wi))$ that is, any move of process i from its state C leads to state W.
- $\bigwedge_{ij}(j < i)AG(C_i U \neg A_j)$ that is, process i can move from its state C to its state W only if every process j which is interconnected with process i and $j < i$ is not in its state A, and such a move is possible some time.

The finite deterministic automata resulted by the method can be seen in figure 2.

Let us notice, that in the case of this synchronization, nothing stops a process from working – that is, stepping in its state W – while the neighbors are working, so in this case, the processes can really work in parallel.

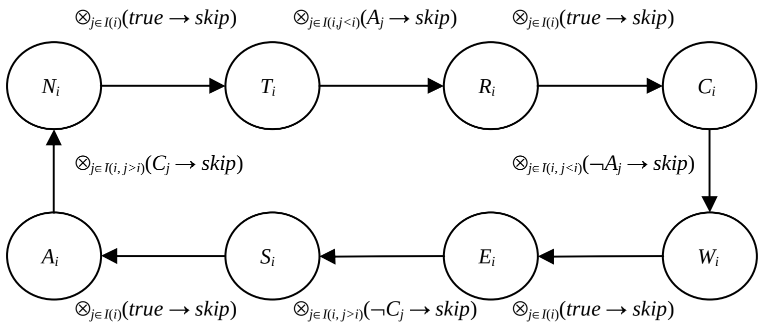

$\otimes_{j\in I(i)}(true \rightarrow skip)$
$\otimes_{j\in I(i,j<i)}(A_j \rightarrow skip)$
$\otimes_{j\in I(i)}(true \rightarrow skip)$
N_i
T_i
R_i
C_i
$\otimes_{j\in I(i,j>i)}(C_j \rightarrow skip)$
$\otimes_{j\in I(i,j<i)}(\neg A_j \rightarrow skip)$
A_i
S_i
E_i
W_i
$\otimes_{j\in I(i)}(true \rightarrow skip)$
$\otimes_{j\in I(i,j>i)}(\neg C_j \rightarrow skip)$
$\otimes_{j\in I(i)}(true \rightarrow skip)$

Figure 2: Improved synchronization skeleton of the embedded system

3.3. The new three-process sytem

The ends of the pipeline must be handled in a special way. At the two ends of the pipeline there are two processes that are connected to only one other process from the pipeline and to a special process. The first process is connected to a sender process and the last process communicates with a receiver process. For the sake of simplicity, let us assume, that the sender process only produces the data and passes on to the first process in the embedded system. Similary, the receiver process only picks up the processed data from the last process in the embedded system and then works with that.

We can look at this set of processes as a system composed by three processes. The first process is the sender, the second is the embedded system and the third is the receiver process. We have to give the synchronization skeleton of this system. This system has only three processes, so we can handle it with the method of Emerson and Clarke [1], without running into the state explosion problem.

There is one more subject, that we have to discuss. The middle process in this system is a system of processes itself, so the specification of our three-process system is quite difficult, because we should not just say for example, that the pseudo process has a state for reading data, because this means in fact, that the first process of the pseudo process reads the data, and at the same time, the last process theoretically can send data, which means that the whole pseudo process sends data to the receiver process, too. This means, that the pseudo process would be in two states at the same time, which is not allowed.

We give two solutions for this issue.

The first solution is that we handle the pseudo process as two processes – and in this case, we have a four-process system instead of a three-process one, of course – which are independent in the four-process system, and one of them is connected to the sender process, and the other is connected to the receiver. This is a reasonable approach, because there is a hidden connection between the two pseudo processes, and this connection is handled by the synchronization of the embedded system.

The second approach is to define the states of the embedded system as pairs, so we will have such state pairs like (N, N), (N, S), (R, S) and so on. For example (N, N) means that the embedded system does not read or send data, while (N, S) means that the system does not read, but sends data and (R, S) means that the system reads and sends data at the same time. With such a type of set of states, we can express the behaviour of the system in a quite efficient way.

The second approach is more complicated than the first one (because of the large number of the states of the system), so we chose the first approach for the solution. We show only the connection between the sender process and the embedded system. The synchronization of the embedded system and the receiver process can be deduced similarly.

The states of the sender process are:

J: normal (working) state,

K: try to send state,

L: sending state,

M: after sending state.

Using these states we can give the temporal logic specification of the system. For the specification, CTL* was used. Based on this specification, we are able to generate the synchronization skeleton of the system. We used the synthetization method of Emerson and Clarke. The synchronization skeleton for the sender process and the first pseudo process of the embedded system can be seen in figure 3. Finally, the synchronization skeleton of the first and the last processes of the embedded system must be modified properly based on the result in figure 3 – the transition conditions of the first process of the embedded system will be the conjunction of the original conditions, and the conditions in the proper transitions of the synchronization of the sender process and the pseudo embedded system – see in figure 4. The new transition conditions for the last process of the embedded system can be deduced similarly.

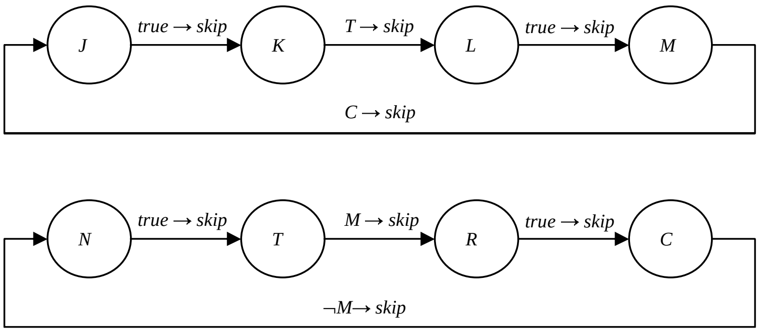
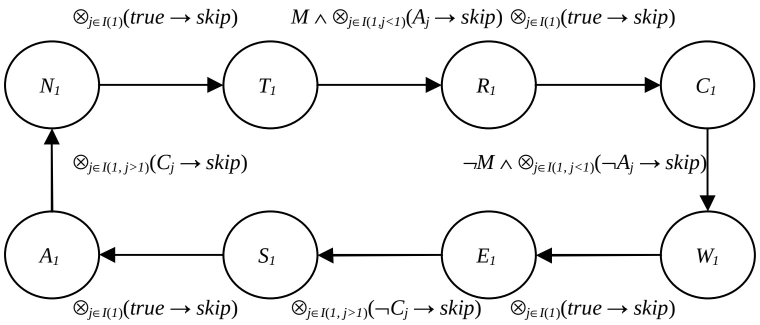


Figure 3: Synchronization skeleton for the sender process and the embedded system



$\otimes_{j\in I(1)}(true \rightarrow skip)$
$M \wedge \otimes_{j\in I(1,j<1)}(A_j \rightarrow skip)$
$\otimes_{j\in I(1)}(true \rightarrow skip)$
N_1
T_1
R_1
C_1
$\otimes_{j\in I(1,j>1)}(C_j \rightarrow skip)$
$\neg M \wedge \otimes_{j\in I(1,j<1)}(\neg A_j \rightarrow skip)$
A_1
S_1
E_1
W_1
$\otimes_{j\in I(1)}(true \rightarrow skip)$
$\otimes_{j\in I(1,j>1)}(\neg C_j \rightarrow skip)$
$\otimes_{j\in I(1)}(true \rightarrow skip)$

Figure 4: The modified synchronization skeleton for the first pocess of the embedded system

4. Future work

The idea for synthetizing the synchronization of pipeline systems comes from hardware designing of graphics cards. We should try to meet these demands.

We should consider the possibilities of using this method in more complicated process or processor networks, for example in the case of hypercube or butterfly networks.

Moreover, a software should be developed, with which the synchronization skeletons can be synthetsized from the specification automatically.

References

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

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

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

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

[5] Z. Hernyák, Z. Horváth, V. Zsók: *Clean-CORBA Interface Supporting Skeletons*, International Conference on Applied Informatics, Eger 2004

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

Postal addresses

Balázs Ugron
Dep. of Software Technology
and Methodology
Eötvös Loránd University
XI. Pázmány P. sét. 1/c.
H-1117 Budapest, Hungary

Szabolcs Hajdara
Dep. of Software Technology
and Methodology
Eötvös Loránd University
XI. Pázmány P. sét. 1/c.
H-1117 Budapest, Hungary