Acta Cybernetica **17** (2005) 123–151.

Synthesis of the synchronization of general pipeline systems *

Balázs Ugron†, Szabolcs Hajdara†, and László Kozma†

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 dealing 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 cannot 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 synthesization method for many similar processes based on a special temporal logic specification.

In this paper we give an extension of this method so that the extended method will be able to handle more properties of parallel systems, especially of pipeline systems. We will consider not only linear [8], but general pipeline systems too. Furthermore, we give an abstract synchronization of a general 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 systems. 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.

*This research work was supported by GVOP-3.2.2-2004-07-005/3.0.

†Department of Software Technology and Methodology, Eötvös Loránd University, Pázmány P. sétány 1/c, H-1117 Budapest, Hungary. E-mail: balee@elte.hu, sleet@elte.hu, kozma@ludens.elte.hu

A pipeline system is a parallel system of processes, which is built in order to solve some kind of problems. In the case of the simplest pipeline system, which is linear, the processes are aligned in 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 is obviously similar, too.

In this paper, we will consider not only linear, but much more general pipeline systems. In paper [8] we described the synthesis of a linear pipeline system, while in this article general pipeline systems are synthesized. We have only the following assumptions:

1. There are some processes, which have only one connection, which is an output connection. These processes generate the data.
2. There are some processes, which have only one connection, which is an input connection. These processes receive the result.
3. The processes inside the pipeline (that is, which are not data generator or receiver processes) are similar in terms of synchronization.

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 cannot be directly applied in this case. For example, the method of Emerson and Clark [2] suffers from the so-called state explosion problem [1], so it cannot be applied for a large number of processes, in practice. Another example is the method of Attie and Emerson [1], 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 [1], 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 review the parts in Attie and Emerson's paper [1] 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 [1].

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 abbreviations for logical disjunction, implication and equivalence can be introduced easily. Furthermore, some additional modalities as abbreviations 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 [1].

2.2 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 $i\ I\ j$ iff processes i and j are interconnected. I is a symmetric and irreflexive relation.

Those process pairs that are in the interconnection relation will be synchronized with each other, while 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 interconnection 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 $\{i_1, \ldots, i_K\}$. $\bigwedge_{ij}$ 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 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 $\forall (i, j) \in I : M, s \models f_{ij}$

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.

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 [2]) can be used to synthesize the synchronization skeleton of an arbitrary process pair from the system.

In this way, 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 synthesized. The method of Attie and Emerson which we roughly described above cannot be applied directly 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 sender and receiver ones of the pipeline are similar, the mentioned two processes differ from them, because they have different state sets from the other processes.

The second reason is that the communication in a pipeline system has a direction – from the sender to the receiver processes. This method does not permit us to distinguish the processes even in the specification, so we cannot handle directions, and the synthesized synchronization code will not be efficient.

First, we give an extension of 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 processes at the ends, handle them as an embedded system, and synthesize the synchronization for them. Finally, we handle the embedded system as a part of a new system, besides the processes at the ends of the pipeline.

3.1 The embedded system

First we give a straightforward solution to 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 enters 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 [1]. The resulted automata can be seen in Figure 1.

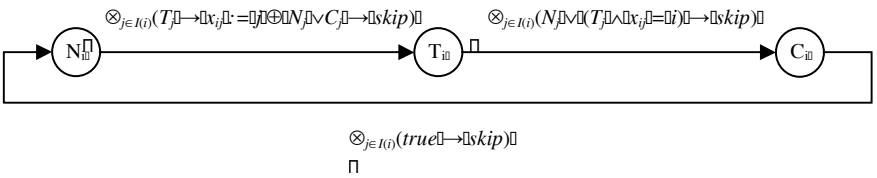

Figure 1: Synchronization skeleton of the embedded system

As we said, this approach is extremely inefficient, because the neighbours of a process cannot 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.

3.2 Another approach for the embedded system

The method introduced in section 3.1 is not really applicable for pipeline systems. In this section, the method of Attie and Emerson will be extended so that it could handle such problems.

We realised that the main problem in the synthesization of this part of the system is that the method does not allow us to make a distinction between processes and we cannot 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 in other words, 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 $\forall i \in \{i_1, \ldots, i_K\} : \rho \to M, s \models f_i$
- $M, s \models \bigwedge_{ij}(\rho) f_{ij}$ iff $\forall (i,j) \in I : \rho \to M, s \models f_{ij}$

This definition intuitively means that a connection between two processes which are 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 synchronization.

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 these states. State N is the start state of every process. State T is a trying state before the R critical section, which 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, which is for sending. Finally, state A is another checkpoint, now after sending.

Let us see the extended MPCTL* specification of the synchronization of the embedded system:

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 N state is into its T state, and such a move is always possible (and similarly for the states R, W and S):

$$\bigwedge_i AG(N_i \Rightarrow (AY_iT_i \wedge EX_iT_i))$$
$$\bigwedge_i AG(R_i \Rightarrow (AY_iC_i \wedge EX_iC_i))$$
$$\bigwedge_i AG(W_i \Rightarrow (AY_iE_i \wedge EX_iE_i))$$
$$\bigwedge_i AG(S_i \Rightarrow (AY_iA_i \wedge EX_iA_i))$$

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

$$\bigwedge_i AG(T_i \Rightarrow AY_iR_i)$$
$$\bigwedge_i AG(C_i \Rightarrow AY_iW_i)$$
$$\bigwedge_i AG(E_i \Rightarrow AY_iS_i)$$
$$\bigwedge_i AG(A_i \Rightarrow AY_iN_i)$$

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

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

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

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

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

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

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

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

$$\bigwedge_i AG(T_i \Rightarrow AFR_i)$$

$$\bigwedge_i AG(C_i \Rightarrow AFW_i)$$

$$\bigwedge_i AG(E_i \Rightarrow AFS_i)$$

$$\bigwedge_i AG(A_i \Rightarrow AFN_i)$$

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

$$\bigwedge_{ij} AG((N_i \Rightarrow AY_jN_i) \wedge (N_j \Rightarrow AY_iN_j))$$

$$\bigwedge_{ij} AG((T_i \Rightarrow AY_jT_i) \wedge (T_j \Rightarrow AY_iT_j))$$

$$\bigwedge_{ij} AG((R_i \Rightarrow AY_jR_i) \wedge (R_j \Rightarrow AY_iR_j))$$

$$\bigwedge_{ij} AG((C_i \Rightarrow AY_jC_i) \wedge (C_j \Rightarrow AY_iC_j))$$

$$\bigwedge_{ij} AG((W_i \Rightarrow AY_jW_i) \wedge (W_j \Rightarrow AY_iW_j))$$

$$\bigwedge_{ij} AG((E_i \Rightarrow AY_jE_i) \wedge (E_j \Rightarrow AY_iE_j))$$

$$\bigwedge_{ij} AG((S_i \Rightarrow AY_jS_i) \wedge (S_j \Rightarrow AY_iS_j))$$

$$\bigwedge_{ij} AG((A_i \Rightarrow AY_jA_i) \wedge (A_j \Rightarrow AY_iA_j))$$

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):

$$\bigwedge_{ij}(j<i)AG((T_i \wedge \neg A_j) \Rightarrow \neg EX_i true$$

$$\bigwedge_{ij}(j<i)AG((C_i \wedge A_j) \Rightarrow \neg EX_i true$$

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

$$\bigwedge_{ij}(i<j)AG((A_i \wedge \neg C_j) \Rightarrow \neg EX_i true$$

8. Always there is a possible step:

$$AGEX true$$

If the set of the process-indices is $\{1,2\}$ (so the processes are P_1 and P_2), then we get the specification of a pair-system. From this specification we can synthesize the synchronization skeleton of the pair-system with the method of Emerson and Clarke [2]. We implicitly applied the method on the parametric spatial operators introduced by us. In the following, the non trivial steps of the synthesis can be seen. Only the main cases are considered, because the other cases can be done by the analogy of the following ones. Note that the dashed lines in the figures mean that trivial steps are omitted there.

Figure 2 shows how the blocks of the initial node can be constructed.

In Figure 3 the construction of the titles of the result of the previous step can be seen.

Figure 4 shows an example of the case when one of the processes has an eventually condition, but because of the parameters of the spatial operators none of the processes has to be blocked.

After this there are a lot of similar steps as Figure 5 shows.

Figure 6 illustrates an example of making blocks of a node where one of the processes has to wait for the another, and Figure 7 shows the titles of the result of this step (only P_1 can execute the changing of its state).

In Figure 8 an example is shown of the case when both processes have the possibility of blocking, but only one of them has to wait for the other. In Figure 9 the titles of the result of the previous step can be seen.

The $\downarrow$ sign in the tableau means that the relevant branch of the tableau is unsatisfiable, so this branch has to be eliminated.

Based on this tableau we can construct the global state transition diagram, which can be seen in Figures 16-20 in Appendix A.

Based on the global state transition diagram, we can construct the DAGs (see Figure 21 in Appendix B) and the fragments (see Figure 22 in Appendix B) of every

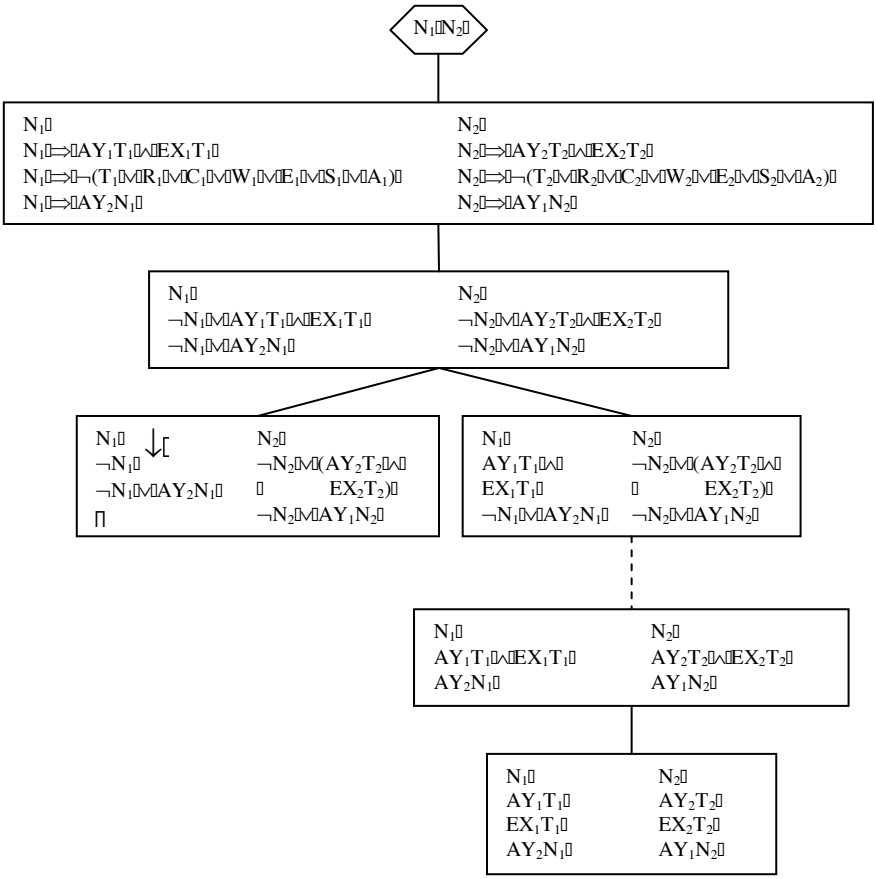

Figure 2: Blocks of the initial node

Figure 3: Titles of the result set of the blocks of the initial node that can be seen in Figure 2

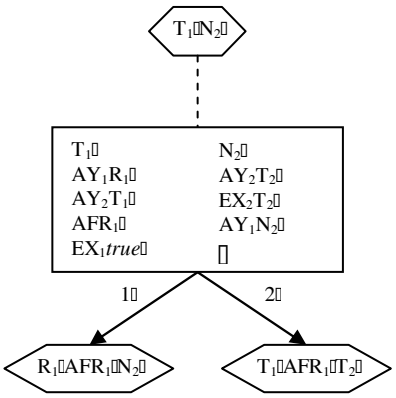

Figure 4: None of the processes blocks but one of them has an eventually condition

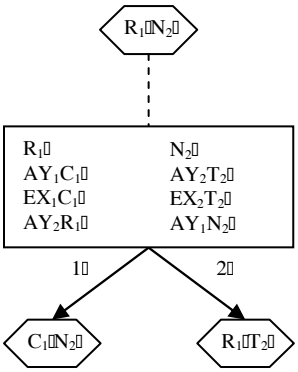

Figure 5: There are a lot of steps similar to step one and two

node. Based on the fragments, the model can be constructed shown in Figure 23-24 in Appendix B.

From the model we can construct the final deterministic automata for all processes. If the first process (P_1) is in its state N_1 then the second process (P_2) can be in all of its states except state R_2, and P_1 has the possibility to step in all of this states. So the condition of the transition of P_1 from state N_1 to state T_1 is $\neg R_2$. The conditions of the transitions in states T_1, R_1, C_1, W_1 are the same. If P_1 is in state E_1 and P_2 is in state C_2, then P_1 cannot step, so the condition of the transition from state E_1 to state S_1 is $\neg R_2 \wedge \neg C_2$. The conditions of the other transitions can be determined similarly. Figure 10 shows the result, the synchronization skeleton for P_1 and P_2.

Note that it cannot happen that P_1 is in state N_1 and P_2 is in state R_2, so the

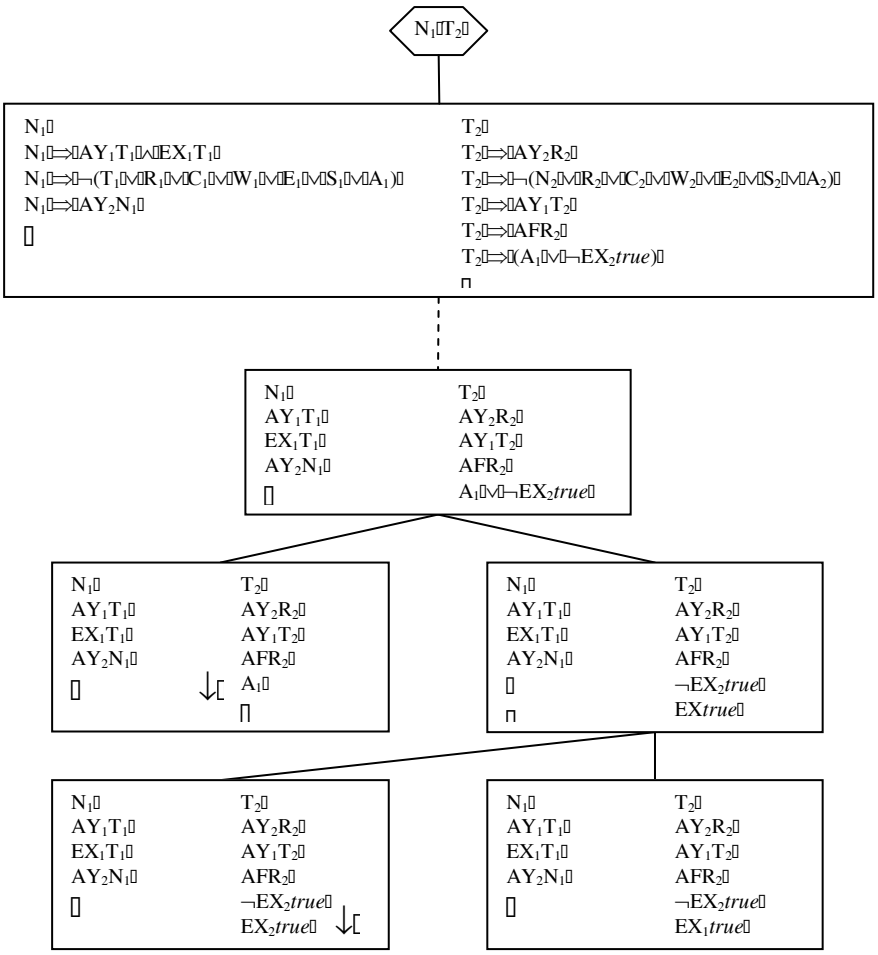

Figure 6: Example of generating blocks of a node where the processes has to wait

$\neg R_2$ in condition of the transition from N_1 to T_1 can be eliminated. Similarly, we can do this with all of the transitions. The simplified synchronization skeletons can be seen in Figure 11.

From this synchronization skeleton we can generate the synchronization code for every process with the method of Attie and Emerson [1]. The finite deterministic automata resulted by the method can be seen in Figure 12.

Note that in the case of this synchronization, nothing keeps a process from working – that is, stepping in its state W – while the neighbours are working, so the processes can really work in parallel in this case.

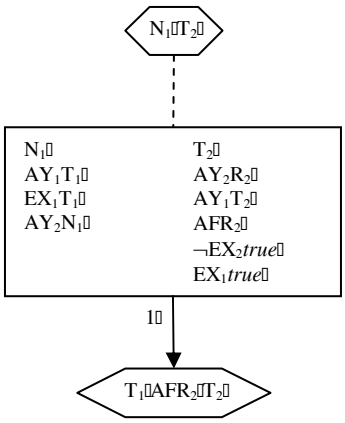

Figure 7: Titles of the result of the previous step can be seen in Figure 6

3.3 The new three-process system

As we said, there are special processes at the two ends of the pipeline, which only send and receive data. For the sake of simplicity, let us assume that the sender processes only produce the data and pass them on to the proper process in the embedded system, and similarly, the receiver processes only pick up the processed data from the proper process in the embedded system, and than work with that. These special processes must be handled in a special way.

The sender and receiver processes are similar in the sense that they are connected with only one another process, which they receive data from, or which they send data to. It is enough to consider only one sender and one receiver process when we generate the synchronization skeleton of the whole system, because the synchronization code of the selected sender and receiver process will naturally be suitable for the other sender and receiver processes, and respectively, the synchronization for the processes that are connected to the sender and receiver processes will be reusable, too. That is why from this point on we will consider only one sender and one receiver process in the system.

Now we can look at our process system as a system composed of three processes. The first process is the selected sender, the second is the embedded system and the third is the selected receiver process. We have to build the synchronization skeleton of this system. This system has only three processes, so we can handle it with the method of Emerson and Clarke [2], without running into the state explosion problem.

There is still one more subject that we have to discuss. The middle process in this system is a system of processes itself, which makes the specification of our three-process system quite difficult. We should not just say, for example, that the pseudo process has a state for reading data, because this means that the first

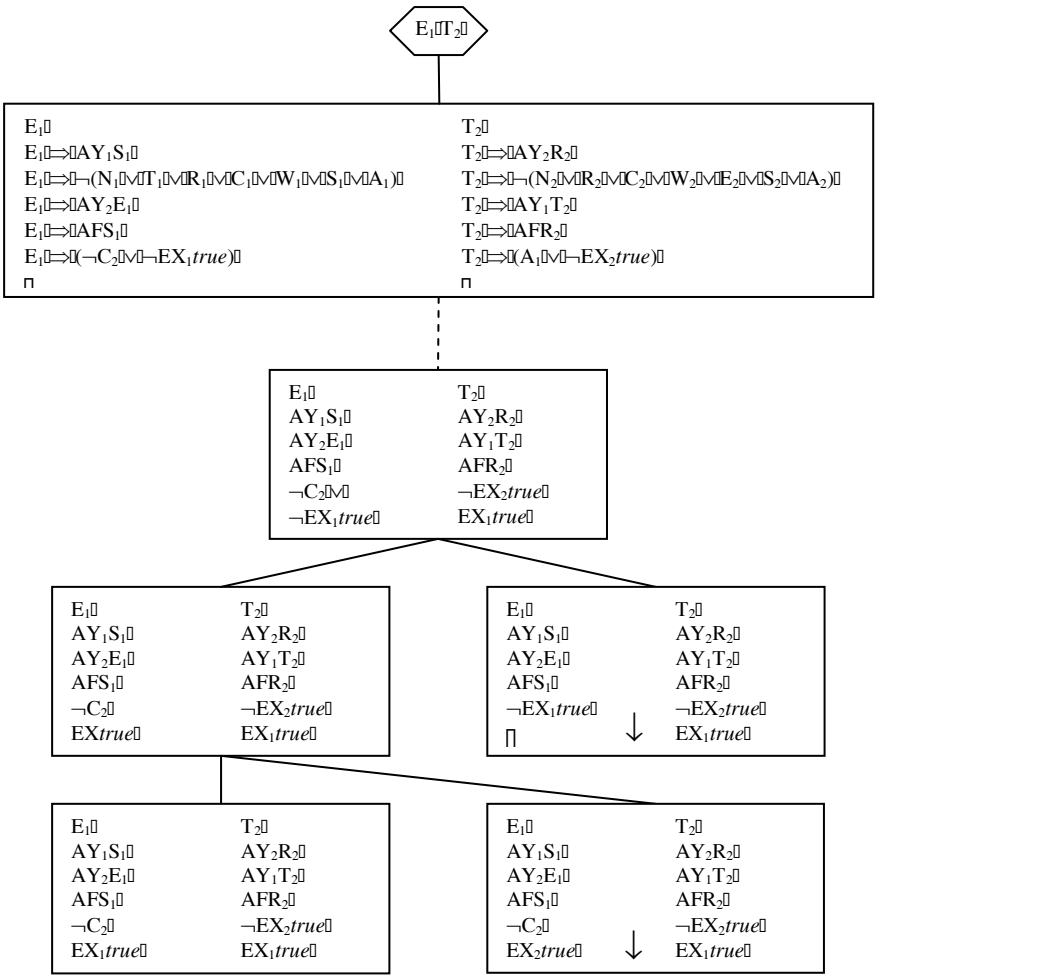

Figure 8: Example of generating blocks of a node where both processes have the possibility to be blocked, but only one of them has to wait

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. As a result, the pseudo process would be in two states at the same time, which is not allowed.

We give two solutions to this issue.

The first solution is that we handle the pseudo process as two processes – in this case, we have a four-process system instead of a three-process one –, which are independent in the four-process system; one of them is connected to the sender

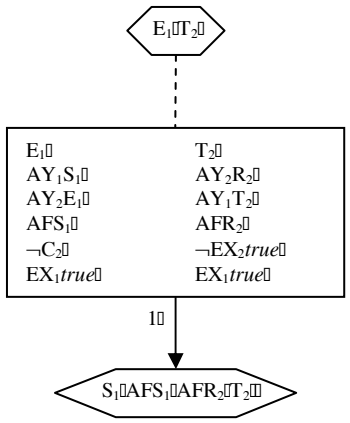

Figure 9: Titles of the result of the previous step can be seen in Figure 7

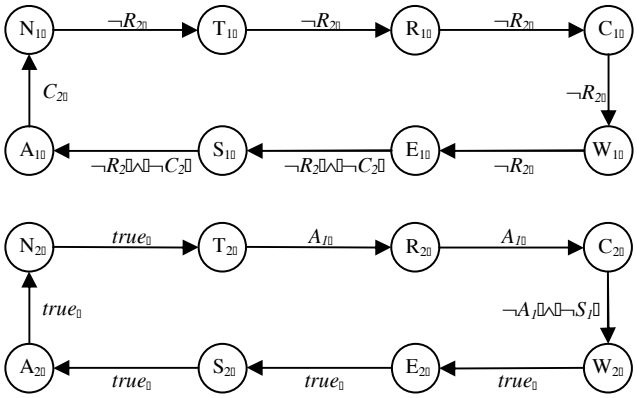

Figure 10: The synchronization skeletons

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 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

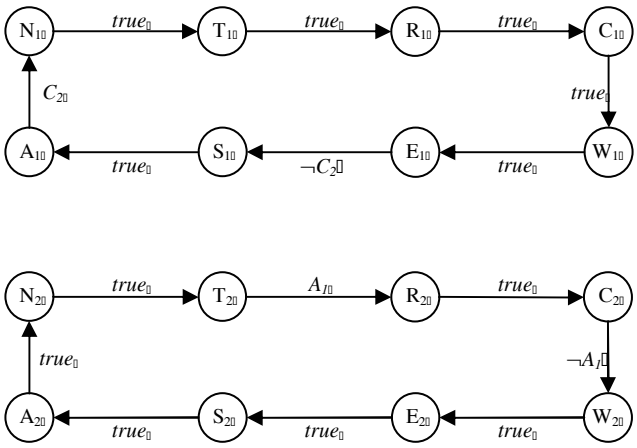

Figure 11: Simplified synchronization skeletons

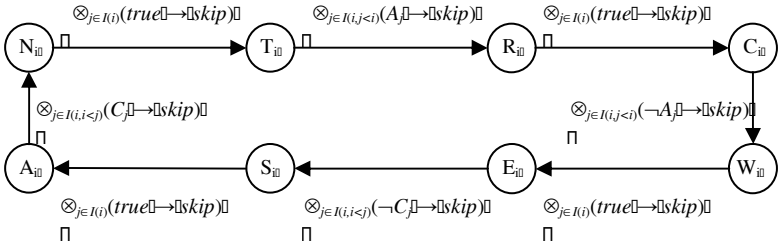

Figure 12: Improved synchronization skeleton of the embedded system

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 – see Appendix C. For the specification, CTL* was used. Based on this specification, we are able to generate the synchronization skeleton of the system. We used the synthesization method of Emerson and Clarke. The synchronization skeleton for the sender process and the first pseudo process of the embedded system can be

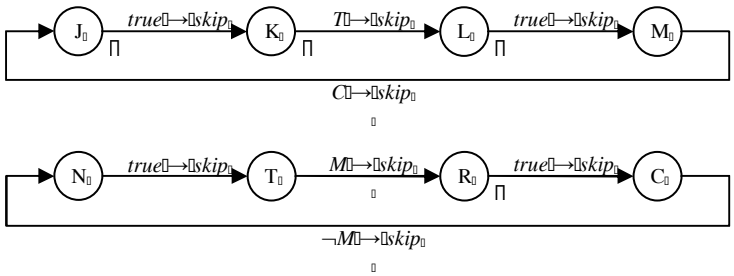

Figure 13: Synchronization skeleton for the sender process and the embedded system

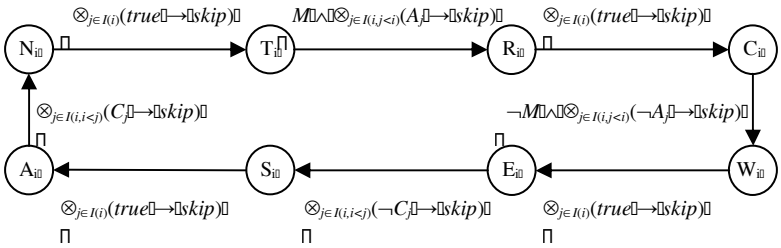

Figure 14: The modified synchronization skeleton for the first process of the embedded system

seen in Figure 13. 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 13 – 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 14. The new transition conditions for the last process of the embedded system can be deduced similarly.

4 An application

Where could this method be used? There are many complicated processor networks, which can be used for computational purposes; for example, the so-called butterfly network (see [7]). An n-level butterfly network can be constructed in a recursive way, which can be seen in Figure 15. The reason why these processor networks are interesting is that there are many parallel algorithms that can be computed on them, such as the Fast Fourier Transformation on a butterfly network (see [7]).

If we apply our result to an n-level butterfly, then we will have n generator processes, which are connected to the nodes at the left side of the butterfly, and we have n receiver processes, which are connected to the nodes at the right side

of the butterfly; the connections between the other processes can be defined in the relation I, in a proper way for the FFT working on a butterfly network. A proper I relation will be described in the following.

The number of the processes in $B_n = n2^{n-1}$.

Let the numbering of the processes in B_2 be like in Figure 15.

Then the numbering of the processes in B_{n+1} comes from the following rules:

- The numbering of the first B_n component in B_{n+1} is the same as of B_n (i.e.: processes $1-4$ of B_3 in Figure 15).
- The numbering of the second B_n component in B_{n+1} is the numbering of B_n shifted by $n2^{n-1}$ (i.e.: processes $5-8$ of B_3 in Figure 15).
- The numbering of the remainder processes (the right side column) is $n2^n + 1 - n2^n + 2^n$ (i.e.: processes $9-12$ of B_3 in Figure 15).

As a result, the relation I consists of the following pairs:

- The pairs in the two B_n components.
- $\forall i \in [1 \dots 2^{n-1}] : ((n-1)2^{n-1} + i, n2^n + i) \in I$ (i.e.: $(3,9)$ and $(4,10)$ of B_3 in Figure 15).
- $\forall i \in [1 \dots 2^{n-1}] : ((n-1)2^{n-1} + i, n2^n + 2^{n-1} + i) \in I$ (i.e.: $(3,11)$ and $(4,12)$ of B_3 in Figure 15).
- $\forall i \in [0 \dots 2^{n-1} - 1] : (n2^n - i, n2^n + 2^{n-1} - i) \in I$ (i.e.: $(7,9)$ and $(8,10)$ of B_3 in Figure 15).
- $\forall i \in [0 \dots 2^{n-1} - 1] : (n2^n - i, n2^n + 2^n - i) \in I$ (i.e.: $(7,11)$ and $(8,12)$ of B_3 in Figure 15).

The synchronization of the communication between the processes are defined in this way, and we do not have to bother with the "business logic" of how the processes compute the data they send to the connected processes.

5 Conclusion

This paper introduced a general pipeline tool, by which a complex application, such as the parallel FFT, can be solved in a short and simple way.

Most of the programs that are working on some kind of data channels (see [5]) can be handled by the method described above. If some modification is still needed, the modification can be restricted to the temporal logic specifications and the relation I, so the above method can be processed by the analogy of the above; moreover, there are different tools exist that can help the process – for instance, the method of Emerson and Clarke [2]; namely, the finite deterministic automata for the pair-system can be generated from the CTL specification automatically, or even

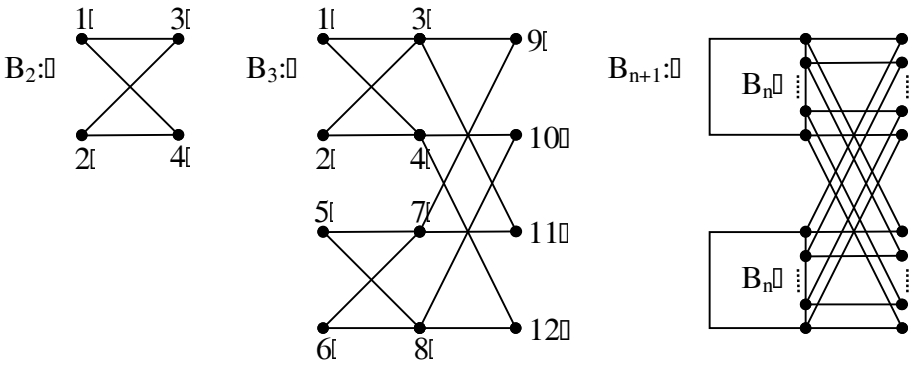

Figure 15: The structure of butterfly processor networks

concrete Java code can be generated with an object-oriented extension (see [4]) of [1] (though a straightforward modification is needed because of the parameterized spatial operators introduced in this paper) etc.

Note, that in the case of FFT no modification was needed, only the proper relation I had to be defined.

6 Future work

The idea of synthesizing the synchronization of pipeline systems comes from the hardware designing of graphics cards. We will work on to meet these demands.

The correctness of the algorithm should be proofed.

An effective tool for deadline checking should be developed.

During the communication, now it is possible that a process has received data but it has to wait until the other processes that receive data from the same sender process receive the data. Theoretically, it would be possible for a data receiver process to step forward in this situation. That is, the data flow control may be improved.

References

[1] 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

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

[3] Sz. Hajdara, L. Kozma, 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*, 6^{th} 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

[7] F. T. Leighton: *Introduction to Parallel Algorithms and Architectures*, 1992

[8] B. Ugron, Sz. Hajdara: *Synthesis of the synchronization of pipeline systems*, 6^{th} International Conference on Applied Informatics, Eger 2004

Appendix A

The following figures (16 – 20) describe the global state transition diagram of the two-process system of the embedded system.

Since the global state diagram of the system is too large, we had to split it into five figures. So we used the following notation: the bold box elements mark those elements that can be continued but they are continued in an other figure. Dotted box elements show boxes which can be found in a previous figure, so in the global state transition diagram there is an edge to such boxes.

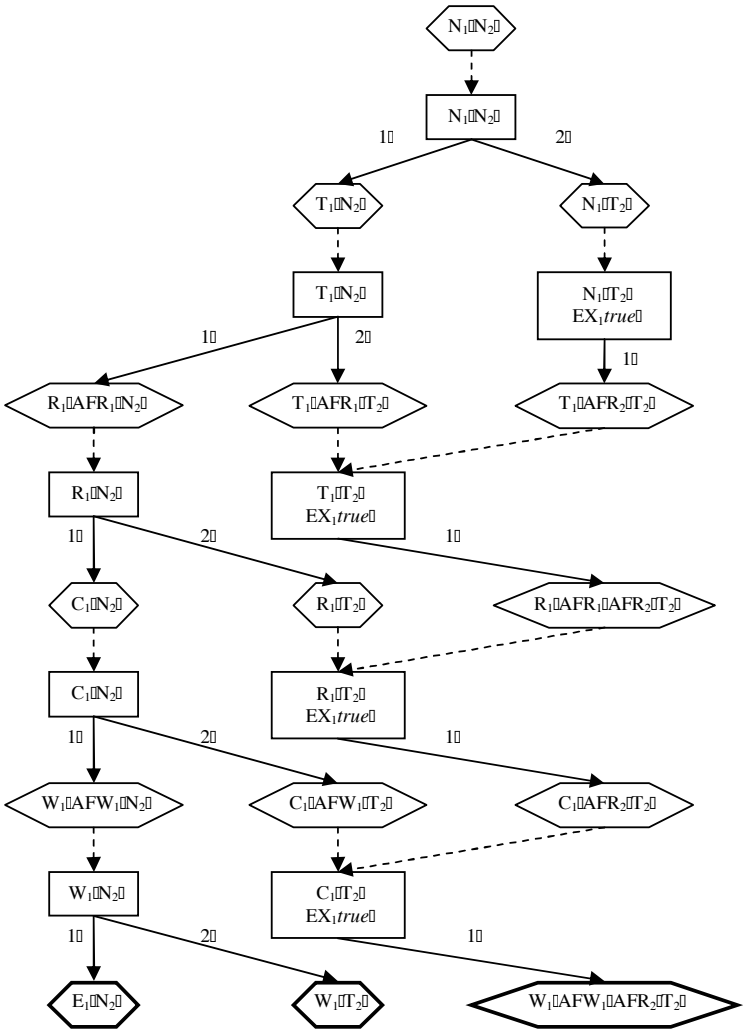

Figure 16: The global state transition diagram (part 1)

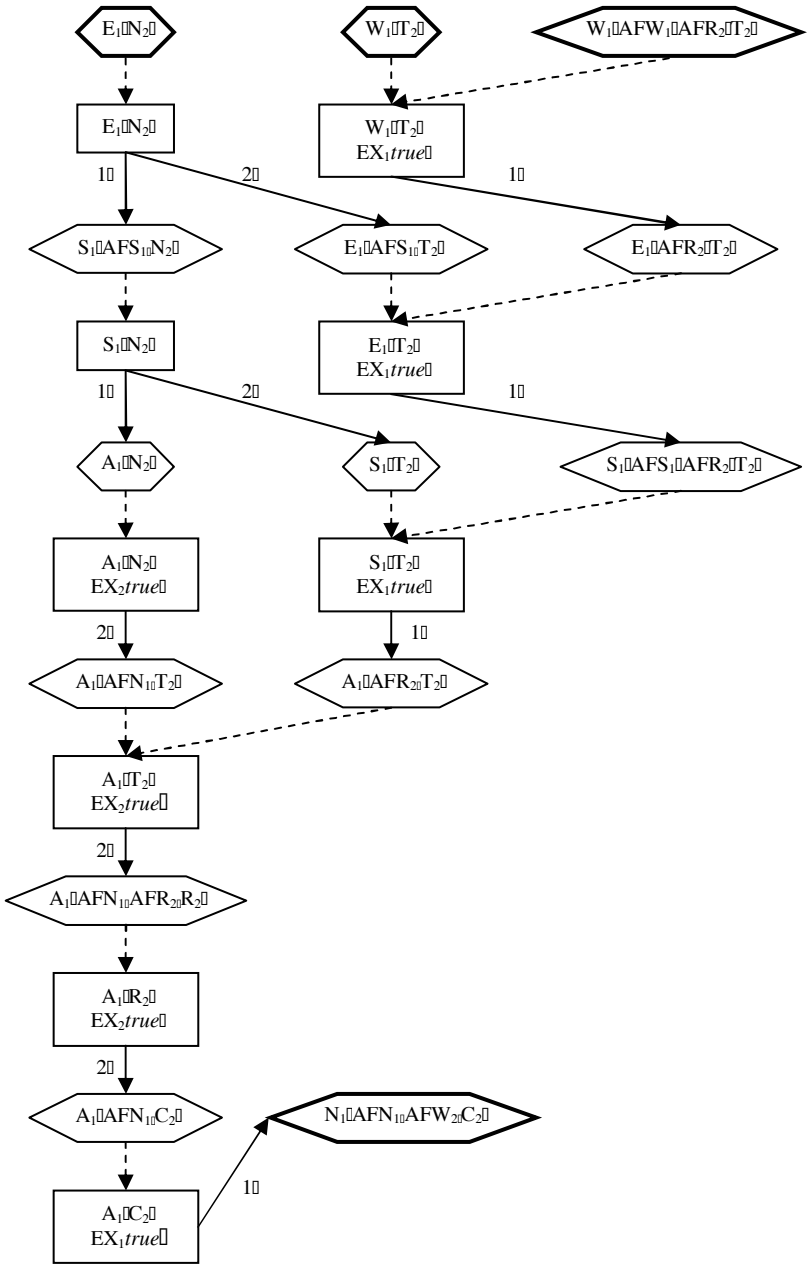

Figure 17: The global state transition diagram (part 2)

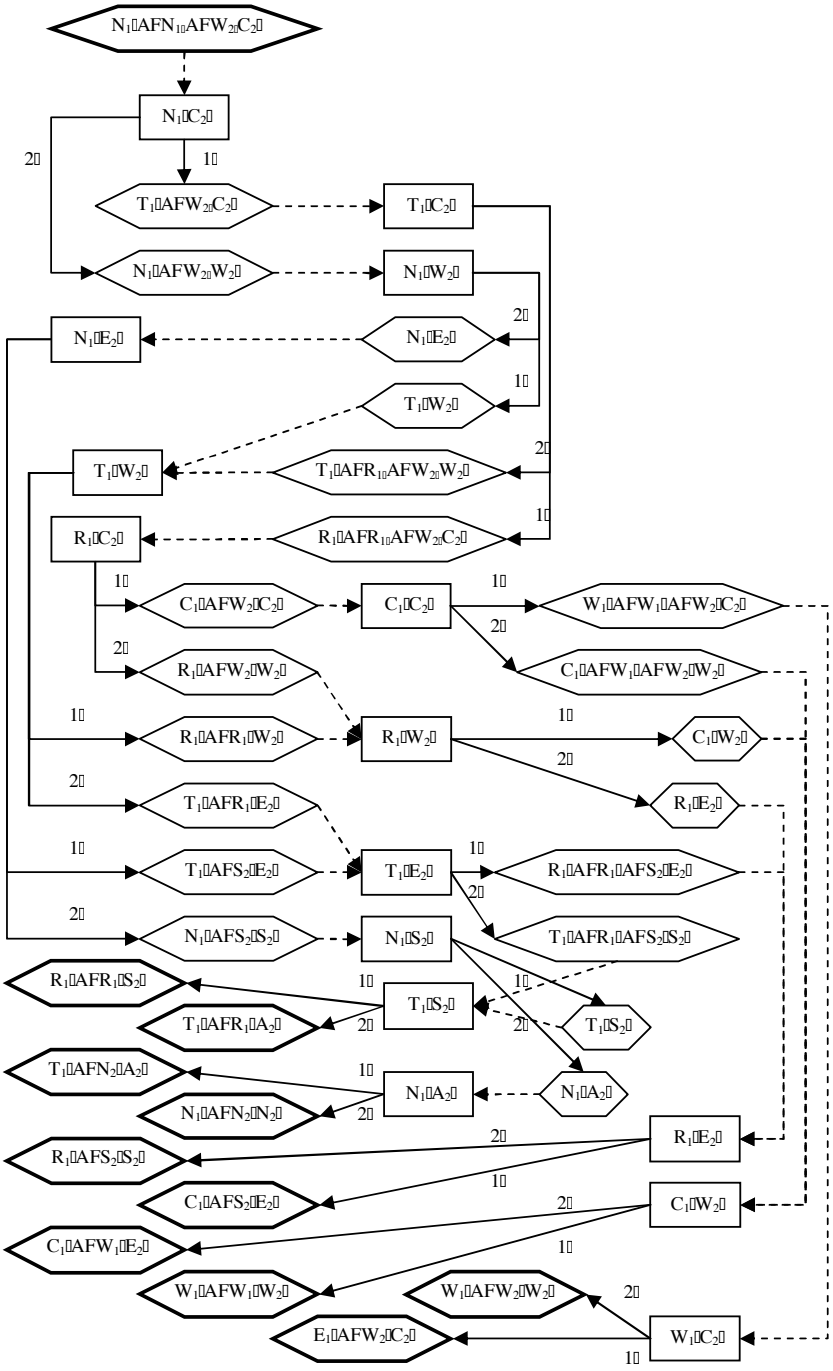

Figure 18: The global state transition diagram (part 3)

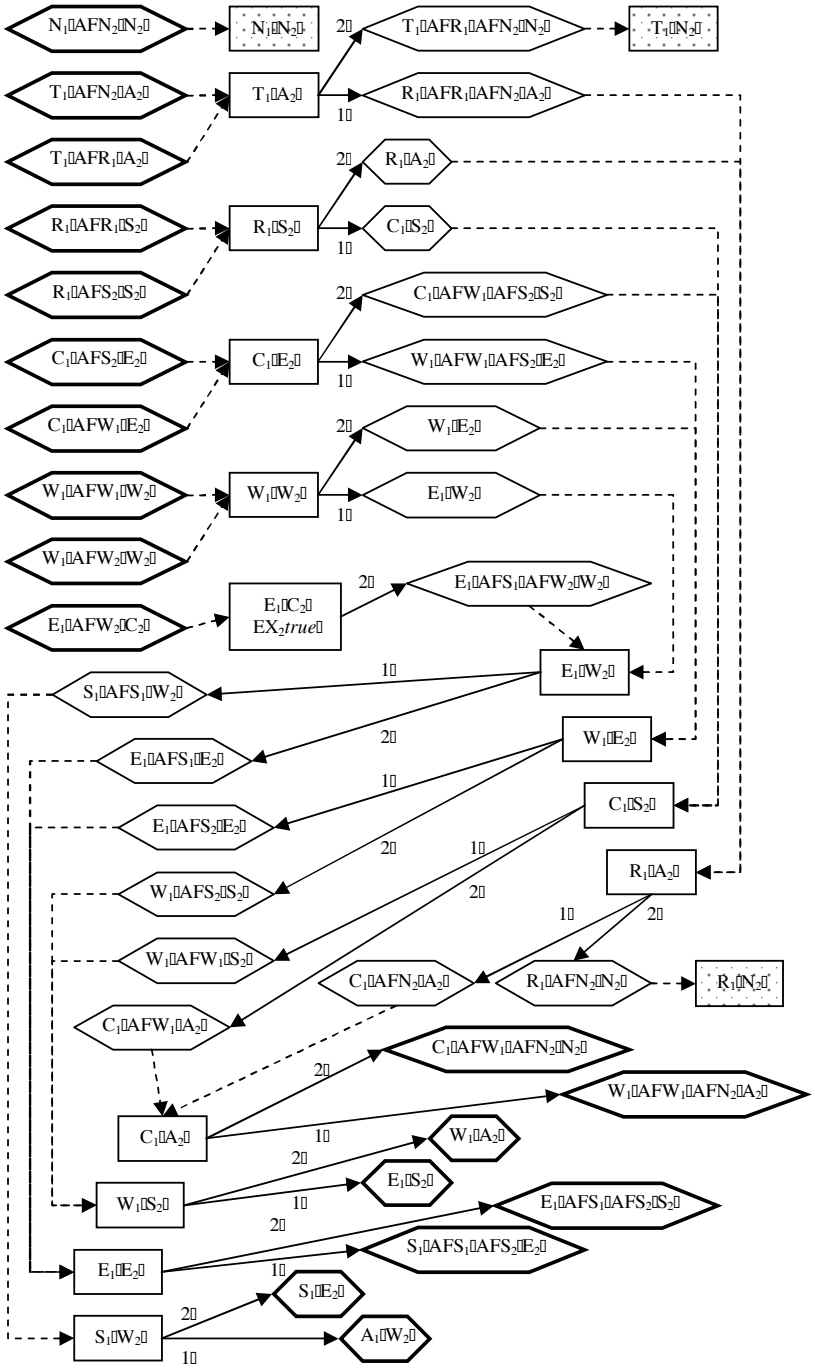

Figure 19: The global state transition diagram (part 4)

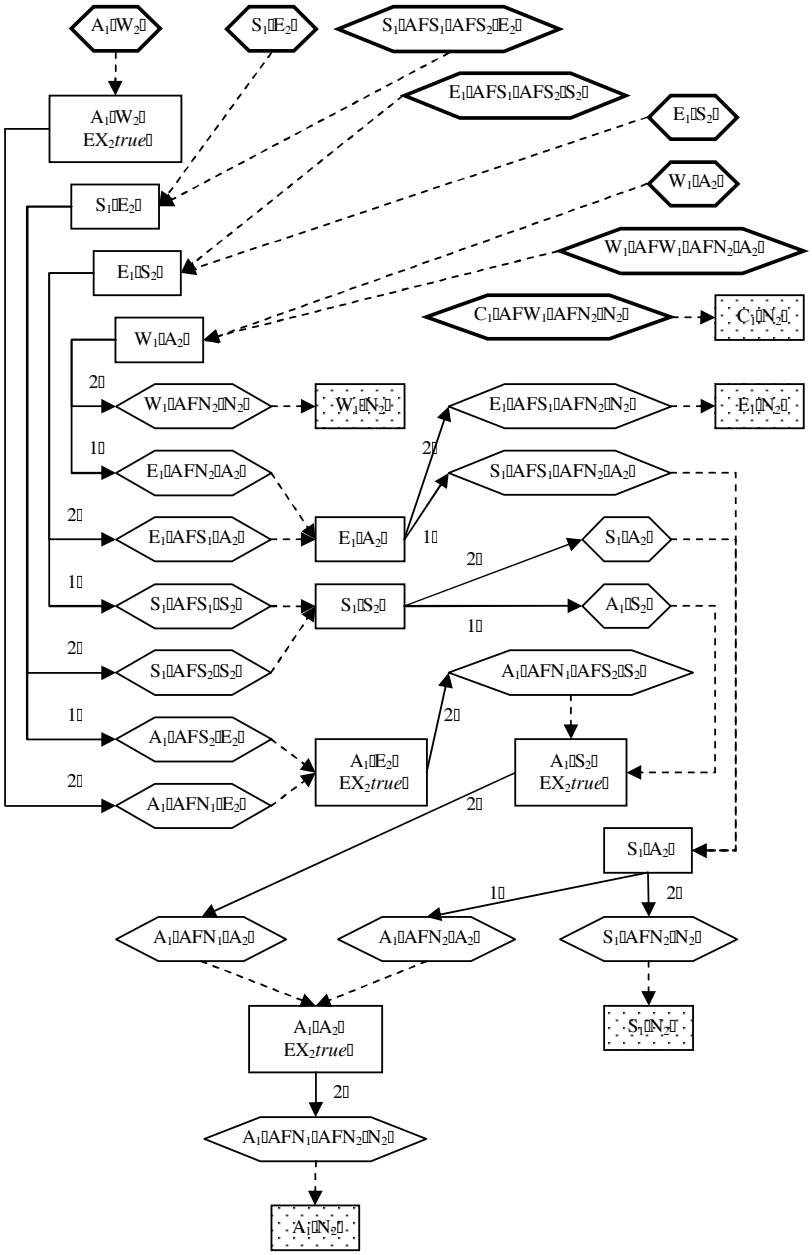

Figure 20: The global state transition diagram (part 5)

Appendix B

The following figures $(21 - 24)$ describe the model of the two-process system of the embedded system.

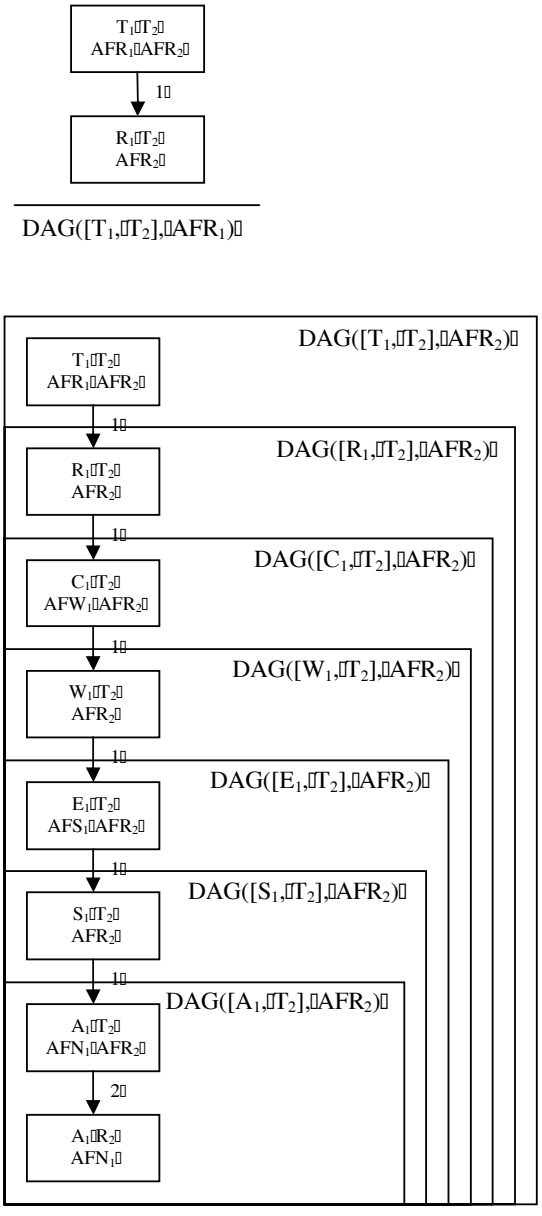

Figure 21: The DAGs that are needed to construct frag$([T_1, T_2])$ □

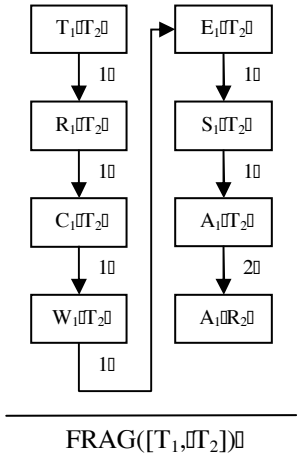

Figure 22: The fragment of the $[T_1, T_2]$ AND node

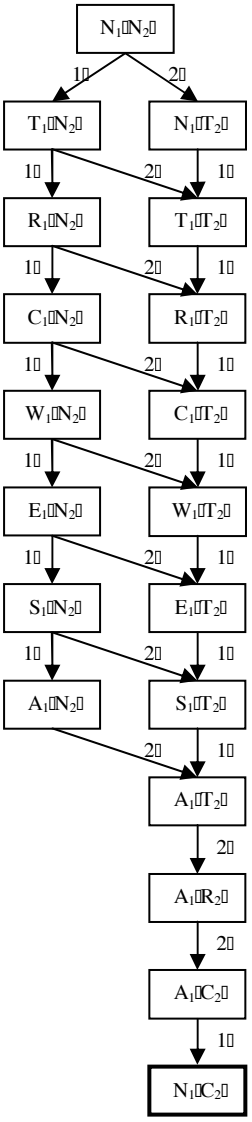

Figure 23: The model of the system

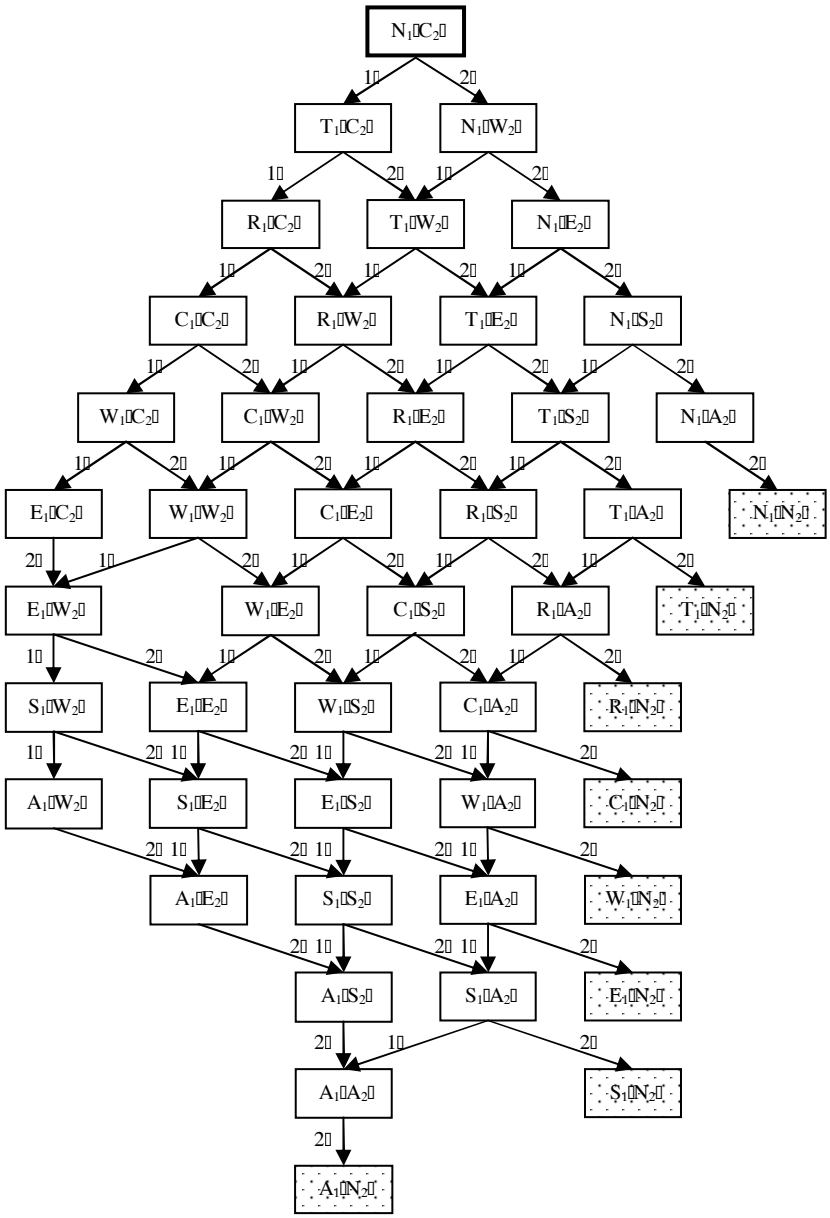

Figure 24: The model of the system

Appendix C

In the following, the temporal logic specification can be seen for the system that is built from the sender process and the embedded system.

For the sake of simplicity, we can join the states N, W, E, S and A of the first process of the embedded system (because the first process only receives data from the sender, the sender will not keep the first process from sending). Let the name of the joined state be N (Normal). Furthermore, we omit the indexes of the states, because the states of the sender process are labeled in another way.

Note that in this case P_1 is the second process and the sender is the first process.

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

$$J \wedge N$$

2. It is always the case that any move P_1 makes from its N state is into its T state, and such a move is always possible (and similarly for the state R and for the states J and L of the sender):

$$AG(J \Rightarrow (AY_1K \wedge EX_1K))$$
$$AG(L \Rightarrow (AY_1M \wedge EX_1M))$$
$$AG(N \Rightarrow (AY_2T \wedge EX_2T))$$
$$AG(R \Rightarrow (AY_2C \wedge EX_2C))$$

3. It is always the case that any move P_1 makes from its T state is into its R state – but such a move is not definitely possible (and similarly for the state C and for the states K and M of the sender):

$$AG(K \Rightarrow AY_1L)$$
$$AG(M \Rightarrow AY_1J)$$
$$AG(T \Rightarrow AY_2R)$$
$$AG(C \Rightarrow AY_2N)$$

4. The processes are always in exactly one state of the state set:

$$AG(J \equiv \neg(K \vee L \vee M))$$
$$AG(K \equiv \neg(J \vee L \vee M))$$
$$AG(L \equiv \neg(J \vee K \vee M))$$
$$AG(M \equiv \neg(J \vee K \vee L))$$
$$AG(N \equiv \neg(T \vee R \vee C))$$

$$AG(T \equiv \neg(N \vee R \vee C))$$
$$AG(R \equiv \neg(N \vee T \vee C))$$
$$AG(C \equiv \neg(N \vee T \vee R))$$

5. Liveness: if P_1 is in state T, then some time it will reach state R (and similarly for the state C and states K and M of the sender):

$$AG(K \Rightarrow AFL)$$
$$AG(M \Rightarrow AFJ)$$
$$AG(T \Rightarrow AFR)$$
$$AG(C \Rightarrow AFN)$$

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

$$AG(J \Rightarrow AY_2J)$$
$$AG(K \Rightarrow AY_2K)$$
$$AG(L \Rightarrow AY_2L)$$
$$AG(M \Rightarrow AY_2M)$$
$$AG(N \Rightarrow AY_1N)$$
$$AG(T \Rightarrow AY_1T)$$
$$AG(R \Rightarrow AY_1R)$$
$$AG(C \Rightarrow AY_1C)$$

7. Data flow control: a process in state T waits for the sender process to reach state M and a process in state C waits for the sender process to leave M (and similarly for the sender):

$$AG((K \wedge \neg T) \Rightarrow \neg EX_1 true$$
$$AG((M \wedge \neg C) \Rightarrow \neg EX_1 true$$
$$AG((T \wedge \neg M) \Rightarrow \neg EX_2 true$$
$$AG((C \wedge M) \Rightarrow \neg EX_2 true$$

8. Always there is a possible step:

$$AGEX true$$

Received November, 2004