Computer Science Journal of Moldova, vol.5, no.3(15), 1997

Design of the Sequential System Automata using Temporal Equivalence Classes

A. Ursu G. Gruita S. Zaporojan

Abstract

A design method of sequential system automata using temporal logic specifications is proposed in this paper. The method is based on well-known Z.Manna and P.Wolper temporal logic satisfiability analysis procedure [1] and is extended to include past time temporal operators. A new specification method which uses temporal equivalence classes is proposed to specify the behaviour of large digital circuits. The impact of the composition and decomposition operations of the temporal equivalence classes on the final automata has been studied. A case study is carried out which deals with the design of the synchronous bus arbiter circuit element. The SMV tool has been used to verify the temporal properties of the obtained automata.

Key words: Temporal Logic, Temporal Equivalence Classes, Temporal Logic Specifications, Sequential Systems, Automata.

1 Introduction

The design and the verification of sequential systems is usually based on finite state automata. Having the finite automaton of a sequential system the designer can implement it using one of the known methods. However the design of the finite automata is usually a difficult problem. A lot of sequential systems have been designed using ad-hock developed finite automata. But this state of art can not satisfy any more the modern circuit design technology. The designer of a large sequential system requires information about the desired functioning of the system. This information is delivered in the form of functional

©1997 by A.Ursu, G.Gruita, S.Zaporojan

specifications which describe the observational behaviour of the system in terms of input/output signals or data. Using this information the designer must built up the finite state automaton of the system. This operation however is less formalised and most tedious.

Temporal logic allows to specify observational behaviour of the systems in terms of input/output signals and histories of signals. To simplify the specification of large circuits the notion of temporal equivalence class is introduced. A temporal equivalence class is a set of input/output signals or histories of input/output signals that cause the same sets of next state input/output signals. Specifying temporal equivalence classes is much simpler with respect to determining all admissible sequences of sequences of input/output signals. Besides, temporal equivalence classes can be introduced as temporal logic formulas, simplifying considerable their analysis and the design of the finite state automaton of the designed system.

The design method proposed in this paper is dedicated to aid the designer to built up the finite state automata of the designed sequential systems using the temporal logic specifications of the system. The method is developed to facilitate the analysis of the temporal logic specifications which describe the desired input/output behaviour of the system and to generate the finite state automaton. The method can be considered in fact as a state identification method since it determines the states of the sequential system. Determining the finite state automaton is important not only for the design process. The verification of a sequential system is usually based on the automaton of the system. Generating the finite state automaton of the designed system the designer can analyse the properties of this automaton and determine whether the initial specifications of the system are correct or not. Moreover the designer usually needs to test an implemented system. To do this the designer may need the automaton of the system. Verifying the automaton of the implemented system with the automaton generated from the temporal logic specifications of the input/output behaviour of the system it is possible to determine whether the implementation is correct or not.

The Z.Manna and P.Wolper procedure [1] for satisfiability analy-

sis of temporal logic specifications is usually utilized to generate the finite state automata of the designed and/or verified systems. This procedure can be used in the design of protocols and real time systems [2]. Such systems can be efficiently specified in terms of future time temporal logic operators. For the design and verification of sequential systems the past time operators are more suitable. To specify a sequential system the designer uses usually timing charts or sequences of input/output vectors [3]. A similar tableau-like procedure for past time temporal logic formulas satisfiability analysis is proposed in the paper. The proposed procedure allows to generate the finite state automata of the sequential systems specified in terms of past time interval temporal formulas. Various equivalent automata can be generated by the method due to inclusion of a relation between temporal equivalence classes. Thus, two or more automata can implement the same temporal specifications of the design system. We call such automata equivalent and they differ by the amount of states they contain. To prove that two automata are equivalent the special study is required. In our paper the equivalence of two automata is considered with respect to the set of important properties. The SMV tool [4] can be efficiently used to check whether an automaton satisfies some properties or not.

2 The design method

The proposed method is a state-based method. This means that the set of possible states of the sequential system is determined by the designer and the admissible sequences of transition between the states are specified using temporal logic formulas. The method is dedicated:

- to facilitate the design of the sequential systems finite state automata;
- to verify the correctness of the implementation of finite state automata;
- to simplify the sequential system automata reducing the amount of states when possible.

As initial data the method uses the input/output relation of the system and the admissible sequences of the input/output (I/O) vectors. The design method consists of the following steps:

1. functional description of the designed sequential system;
2. design of the temporal logic specifications which describe the input/output behaviour of the system using temporal equivalence classes;
3. satisfiability analysis of the temporal logic specifications;
4. design of the finite state-graph of the specifications;
5. design of the finite automaton of the specifications;
6. verification of the finite automaton using finite state machine verification tools.

All the steps of the method will be explained properly in the example below. Here we would like to mention that the satisfiability analysis of the specifications (step 3) is based on an algorithm like the Z.Manna and P.Wolper procedure [1] but is extended by authors to include the analysis of the past time intervals formulas [3].

2.1 The past time formulas satisfiability analysis algorithm

The temporal logic specifications of sequential systems include usually formulas of the form

$$[](f_1 \supset \bigcirc(\neg Process\, U(f_2 \vee f_3 \vee ... \vee f_n))),$$

where the formula f_1 specifies a state or an event of the system and the formula $(\neg Process\, U(f_2 \vee f_3 \vee ... \vee f_n))$ specifies the next time state or event which must be one described by a formula of the list $(f_2, f_3, ..., f_n)$. If the formulas $(f_1, f_2, f_3, ..., f_n)$ are present or future time formulas (formulas which do not contain past time temporal operators) the satisfiability analysis can be performed using the Z.Manna

and P.Wolper procedure [1]. The future time formulas are very popular when specifying sequential system automata for design purposes. The difficulties appear when the formulas $(f_1, f_2, f_3, ..., f_n)$ are past time formulas. The past time formulas are very popular when specifying timing charts generated by sequential circuits for testing purposes [3] or when specifying the admissible sequences of input/output vectors of such systems. The Z.Manna and P.Wolper procedure does not work efficiently for these type of formulas which include past time operators.

To perform the satisfiability analysis of the temporal logic specifications which include past time formulas we introduce a tableau-like algorithm which allows to analyse the past time intervals formulas of the form

$$[]((f_1 S f_2 S ... S f_m) \supset \bigcirc(\neg Process\, U(f_{m+1} \vee f_{m+2} \vee ... \vee f_n))),$$

and

$$[]((f_1 \beta f_2 \beta ... \beta f_m) \supset \bigcirc(\neg Process\, U(f_{m+1} \vee f_{m+2} \vee ... \vee f_n))).$$

The expression $(f_1 S f_2 S ... S f_m)$ denotes the $(m-1)$-interval history of the process specified in the chronological order by the formulas $f_m, f_{m-1}, f_{m-2}, ..., f_2, f_1$. We call such formulas $(m-1)$-interval history formulas. The expression $(f_1 \beta f_2 \beta ... \beta f_m)$ denotes the $(m-1)$ or less interval of history of the process specified in the chronological order by the formulas $f_1; f_2, f_1; ..., f_m, f_{m-1}, f_{m-2}, ..., f_2, f_1$. We call such formulas back-to $(m-1)$-intervals history formulas. The proposed past time formulas satisfiability algorithm consists of the following steps:

1. The initial set of formulas Σ_0 forms the root node of a tree structure.

2. Perform the Z.Manna and P.Wolper satisfiability analysis of the initial set of formulas Σ_0. If the formulas are not satisfiable go to step 5, otherwise generate the next time sets of decomposed formulas $(\Sigma_0^1, \Sigma_0^2, \Sigma_0^3, \ldots, \Sigma_0^n)$. These sets form the descendent nodes with respect to the node Σ_0. The edges which link the initial node Σ_0 with the nodes Σ_0^i, $(i = 1 \ldots n)$ are labelled by

the propositional variables $f_{l_i}(l_i = 1 \ldots n)$ which satisfy the sets $\Sigma_0^i, (i = 1 \ldots n)$ respectively. The nodes containing the sets already present in the tree are considered terminal nodes. If all leafs of the tree are terminal nodes go to step 4, otherwise go to step 3.

3. For an arbitrary nonterminal node of the tree, Σ_j^i, perform the Z.Manna and P.Wolper satisfiability analysis taking into account the history of the node. The history of a node is the path from the tree root to the current node. The path is formed of the propositional variables $f_k, (k = 1 \ldots p)$ on the path edges. If the formulas corresponding to the current node are not satisfiable go to step 5. If no nonterminal node has been generated go to step 4, otherwise go to step 3.

4. The satisfiability analysis of specifications has been performed successfully. The transitive closure of the sets of decomposed formulas has been generated. Stop

5. The specifications are not correct. Stop.

3 Design of a sequential system: A case study

As a case study the design of a sequential system of a synchronous bus arbiter circuit is presented in this section. The example has been used in [4] to verify its correctness using CTL and SMV tool. In our paper we treat the synthesis problem. Our goal is to generate the finite-state automaton of the bus arbiter element provided the temporal logic specifications of the input/output vectors sequences are given. As described in [4] the purpose of the bus arbiter is to grant access on each clock cycle to a single client among a number of k clients contending for the use of a bus (or other resource). The inputs to the circuit are a set of request signals $req_0 \ldots req_{k-1}$ and the outputs are a set of acknowledge signals $ack_0 \ldots ack_{k-1}$. Normally, the arbiter asserts the acknowledge signal of the requesting client with the lowest index. However, as requests become more frequent, the arbiter is designed to

fall back on a round robin scheme, so that every requester is eventually acknowledged. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every clock cycle. If a given client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus.

The bus arbier consists of k arbiter elements. Each arbiter element services one input request line. Since the arbiter elements are identical we will consider a single arbiter element. According to the functional description of the bus arbiter an arbiter circuit element has four inputs (*Request, Token_in, Override_in, Grant_in*) and four outputs (*Ack, Token_out, Override_out, Grant_out*) [4]. The input and output signals of the arbiter element are presented in Fig. 1.

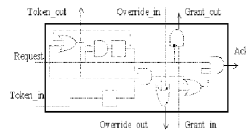

Fig.1. The intended black box (of the synchronous bus arbiter element)

The black box in Fig.1. is an intended black box since its internal structure is shown by dotted lines and it corresponds to the synchronous bus arbiter circuit element as in [4]. We give the internal structure of the arbiter element to simplify the understanding of the proposed method.

3.1 Functional specifications of the designed system

To specify the system the input/output relation of the system signals must be introduced. The relation between inputs and outputs of the synchronous bus arbiter circuit element can be introduced using I/O vectors. An I/O vector is a tuple of 8 binary values, consisting of 4 input and 4 output elements signals as follows: (*Request, Token_in, Override_in, Grant_in, Ack, Token_out, Override_out, Grant_out*). The relation between inputs and outputs can not be represented as a simple functional dependence. The outputs depend not only on the input signals but on the history of the input sequence also. That is why the temporal aspect must be taken into account. Thus, to generate the input/output relation the designer has to:

1. determine all possible values of input/output vectors;
2. specify all possible sequences of input/output vectors.

The set of all possible I/O vectors can be introduced by the designer as different desired or observed reactions of the system to different inputs. This vectors are introduced as a truth table (Table 1).

Table 1

The synchronous bus arbiter element I/O vectors

Signals	I/O vectors									
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Request	0	0	0	0	0	0	0	0	1	1
Token_in	0	0	0	0	1	1	1	1	0	0
Override_in	0	0	1	1	0	0	1	1	0	0
Grant_in	0	1	0	1	0	1	0	1	0	1
Ask	0	0	0	0	0	0	0	0	0	1
Token_out	0	0	0	0	1	1	1	1	0	0
Override_out	0	0	1	1	0	0	1	1	0	0
Grant_out	0	1	0	1	0	1	0	1	0	0

Table 1 (continued)

The synchronous bus arbiter element I/O vectors

Signals	I/O vectors								
	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}
Request	1	1	1	1	1	1	1	1	1
Token_in	0	0	1	1	1	1	1	1	1
Override_in	1	1	0	0	1	1	0	0	1
Grant_in	0	1	0	1	0	1	0	1	0
Ask	0	1	0	1	0	1	1	1	1
Token_out	0	0	1	1	1	1	1	1	1
Override_out	1	1	0	0	1	1	1	1	1
Grant_out	0	0	0	0	0	0	0	0	0

From the Table 1 it follows that the system is sequential due to the vectors that have the same input and different output parts. For example, the vectors (f_{13}, f_{17}), (f_{14}, f_{18}), (f_{15}, f_{19}) have the same input but different output parts. This situation is possible only when the system acts on different states. Each pair of mentioned above pairs of I/O vectors require two states since any two vectors with the same input but with different output parts can be implemented only in different

states. On the other hand, since there are three pairs of such vectors and supposing that these states are all different the maximal number of different states can be determined as $2 + 2 + 2 = 6$. Thus, in this example the system may have from two up to six states. That is why the set of all possible values of input/output vectors is not sufficient to specify the system. Additional information about the system states is required. This information can not be introduced explicitly since Table 1 does not contain such information. Identification of the automaton states is the scope in our design. As identification information we use the admissible sequences of I/O vectors. These sequences must represent all possible "histories of the system". Liner time temporal logic formulas are used to specify the set of all possible sequences of input/output vectors.

3.2 Temporal Logic Specifications of the system

To specify the set of all possible sequences of I/O vectors the designer must know them. Sometimes it is impossible to know all sequences of I/O vectors. Moreover, even in the case it is possible to list all admissible sequences of I/O vectors, this information is of less use. The designer has to transform them into an automaton but it is not so simple since the number of all admissible sequences can be very large or infinite. In this paper Liner Time Temporal Logic is used to specify all admissible sequences of I/O vectors. To do this the temporal equivalence classes are used.

A temporal equivalence class is a set of I/O vectors or I/O vector histories that cause the same sets of next state I/O vectors. To specify temporal equivalence classes is much simpler with respect to determining all admissible sequences of I/O vectors. Besides, temporal equivalence classes can be introduced as temporal logic formulas, simplifying considerable their analysis and the generation of the finite state automaton of the designed system.

For example, the functional analysis of the arbiter element allows to determine that the vectors f_1, f_2, f_3, f_4 belong to a temporal equivalence class since the sets of next state vectors caused by each of these

vectors correspond to $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}$.

Using Linear time Temporal Logic the relation corresponding to this class of temporal equivalence can be represented as follows,

$$[](f_1 \vee f_2 \vee f_3 \vee f_4) \supset \bigcirc(\neg Process\, U(f_1 \vee f_2 \vee f_3 \vee f_4 \vee f_5 \vee f_6 \vee f_7 \vee f_8 \vee f_9 \vee f_{10} \vee f_{11} \vee f_{12} \vee f_{13} \vee f_{14} \vee f_{15} \vee f_{16}))),$$

or simply

$$[](\bigvee_{i=1}^{4} f_i) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{16} f_i))).$$

A rigorous functional analysis of the designed synchronous bus arbiter element allows to specify four temporal equivalence classes as specified by (1), (2), (3) and (4).

$$[]((\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12, j=1..4} (f_i \beta f_j)) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{16} f_i))); \qquad (1)$$

$$[]((\bigvee_{i=5}^{8} f_i \bigvee_{i=13}^{15} f_i \vee f_{16} \wedge \bigodot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12, j=1..4} (f_i \beta f_j))) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i))); \qquad (2)$$

$$[]((\bigvee_{i=17}^{19} f_i \vee f_{16} \wedge \neg \bigodot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12, j=1..4} (f_i \beta f_j))) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i))); \qquad (3)$$

$$[]((\bigvee_{i=9..12, j=5..19} (f_i S f_j)) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i))); \qquad (4)$$

$$[]((\bigvee_{i=1}^{19} f_i) \wedge \neg(\bigwedge_{i<i<j<19} (f_i \wedge f_j))); \qquad (5)$$

$$Process \equiv (\bigvee_{i=1}^{19} f_i); \quad (6)$$

$$\neg Process\, U(\bigvee_{i=1}^{16} f_i); \quad (7)$$

The formula (5) specifies the single event condition. It means that a single I/O vector can be processed at a time. The formula (6) defines the variable $Process$. This variable allows to specify the current arbiter element. A more complex specification of the bus arbiter could use the $Process$ variable to identify the arbiter element.

In these specifications the following temporal operators have been used [1]: $[] -$"$Always$", $\bigcirc -$"$Next$", $\odot -$"$Prevoiusly$", $U -$"$Until$", $S -$"$Since$", $\beta -$"$Back - to$". We would like to underline that the formulas in the specifications include the "$Since$" and "$Back - to$" temporal operators S and β. These operators specify the behaviour of the system in the past. So the expression $\odot f_i$ tells that in the previous state f_i happened. The expression $(f_i S f_j)$ specifies that for the current state in which f_i happened there was a state in the past in which f_j happened and no other f_k happened between these two states except may be a continuous sequence of f_i. Respectively the expression $f_i \beta f_j$ specifies that for the current state in which f_i happened there was a state in the past in which f_j happened and no other f_k happened between these two states except may be a continuous sequence of f_i, or no state exists in the past in which f_j happened and then f_i happened so far.

3.3 Satisfiability analysis of the specifications

For the satisfiability analysis of the temporal logic specifications a tableau-like method has been developed which allows to perform the analysis of the specifications which include future time operators $\bigcirc, U, []$ - as well as past time temporal logic operators $\odot, S$ and β. First, performing the satisfiability analysis the following additional formula has been generated:

$$\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i). \qquad (8)$$

The satisfiability analysis procedure allows to determine the transitive closure of the temporal logic formula derived from the initial set of formulas. This transitive closure consists of a set of sets of temporal logic formulas valid under the assumption that they are valid on the future or/and on the past. The future and past are presented in these formulas by temporal logic operators. The results of satisfiability analysis using the proposed method are presented in Table 2. From the table it follows that only two sets of formulas Σ_0 and Σ_1 have been created during the analysis. This sets represent two different states of the arbiter element.

Table 2

The satisfiability analysis tableau

Nr	Formula set	Proposition	Formulas		
1	2	3	4	5	6
			$f_1, f_2,$ f_3, f_4	$f_5, f_6,$ f_7, f_8	$f_9, f_{10},$ f_{11}, f_{12}
1	Σ_0		$1 \div 7 \equiv \Sigma_0$	$1 \div 6, 8 \equiv \Sigma_1$	$1 \div 7 \equiv \Sigma_0$
2	Σ_1	$f_5 \div f_8,$ $f_{13} \div f_{16}$	$1 \div 7 \equiv \Sigma_0$	$\emptyset$	$1 \div 6, 8 \equiv \Sigma_1$
			7	8	9
			$f_{13}, f_{14},$ f_{15}	f_{16}	$f_{17}, f_{18},$ f_{19}
1	Σ_0		$1 \div 6, 8 \equiv \Sigma_1$	$1 \div 6, 8 \equiv \Sigma_1$	$\emptyset$
2	Σ_1	$f_5 \div f_8,$ $f_{13} \div f_{16}$	$1 \div 6, 8 \equiv \Sigma_1$	$1 \div 6, 8 \equiv \Sigma_1$	$1 \div 6, 8 \equiv \Sigma_1$

3.4 Design of the finite state automaton of the specifications

In sequential system design it is very important to built up the automata of the designed systems. The satisfiability analysis performed above (Table 2) allows to built up the state-graph of the specifications. Below we present the state-graph of the specifications (Fig.2).

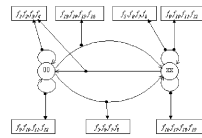

Fig.2. The 2-states automaton of the synchronous bus arbiter element.

In the Fig.2 the transitions between the only two states are marked by the I/O vectors which cause them. As we see the automaton of the designed synchronous bus arbiter element consists of 2 states. Hence, the pairs or I/O vectors $(f_{13}, f_{17}), (f_{14}, f_{18}), (f_{15}, f_{19})$ which have the same input but different output parts are caused by two different states. Namely, the vectors (f_{13}, f_{14}, f_{15}) are specific for the state 00 while the vectors (f_{17}, f_{18}, f_{19}) are specific for the state XX.

From the automaton in Fig.2 it follows that there exists only two temporal equivalence classes. A node of the automaton corresponds to a temporal equivalence class because all the vectors entering the node cause the same set of next time vectors. Indeed the equivalence classes

specified by (2), (3) and (4) belong to the same equivalence class which is the union of these classes. So the formulas (2), (3), (4) can replaced by the formula:

$$[](((\bigvee_{i=5}^{8} f_i \bigvee_{i=13}^{15} f_i \vee f_{16} \wedge \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j)))\vee$$
$$(\bigvee_{i=17}^{19} f_i \vee f_{16} \wedge \neg \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j)))\vee$$
$$(\bigvee_{i=9..12,j=5..19} (f_i S f_j))) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i))),$$

which can be simplified to

$$[](((\bigvee_{i=5}^{8} f_i \bigvee_{i=13}^{19} f_i \vee (\bigvee_{i=9..12,j=5..19} (f_i S f_j))) \supset nonumber \quad (9)$$
$$\bigcirc(\neg Process\, U(\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i))). \quad (10)$$

3.5 Decomposing temporal equivalence classes

We have seen that sometimes it is possible to construct major temporal equivalence classes using minor ones. We call this operation the composition of temporal equivalence classes. The reverse operation, the decomposition, can also be used to get desired design solutions. This operation is important when the user wants to obtain an automaton specifying explicitly the states he expects to obtain. Namely, suppose in our example it is necessary to obtain an automaton with 4 different states as specified by four temporal equivalence classes (1), (2), (3) and (4). Unfortunately the satisfiability analysis algorithm allows to generate a 2-state automaton (Fig.2). This is because the formulas (2), (3) and (4) represent subclasses of a common temporal equivalence class. The algorithm generates separate states only for major classes. To generate separate states for minor classes we have to separate them in such a way that their union could not produce a major class. The simplest way to do this is to include the antecedent of a temporal equivalence

class formula into its consequent part via conjunction. And since the consequent is a next time formula the included antecedent must be prefixed by a "*Previously*" operator to keep the meaning of the temporal equivalence class formula. This operation is needed for the satisfiability analysis algorithm and is equivalent to including the history of the process into the next time formulas generated by the algorithm. Now the formulas (1), (2), (3) and (4) can be rewritten as:

$$[]((\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j)) \supset \bigcirc(\neg Process\, U(\bigvee_{i=1}^{16} f_i))); \tag{1'}$$

$$[]((\bigvee_{i=5}^{8} f_i \bigvee_{i=13}^{15} f_i \vee f_{16} \wedge \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j))) \supset \bigcirc(\neg Process\, U(\odot(\bigvee_{i=5}^{8} f_i \bigvee_{i=13}^{15} f_i \vee f_{16} \wedge \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j))) \wedge (\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i)))); \tag{2'}$$

$$[]((\bigvee_{i=17}^{19} f_i \vee f_{16} \wedge \neg \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j))) \supset \bigcirc(\neg Process\, U(\odot(\bigvee_{i=17}^{19} f_i \vee f_{16} \wedge \neg \odot(\bigvee_{i=1}^{4} f_i \bigvee_{i=9..12,j=1..4} (f_i \beta f_j))) \wedge (\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i)))); \tag{3'}$$

$$[]((\bigvee_{i=9..12,j=5..19} (f_i S f_j)) \supset \bigcirc(\neg Process\, U(\odot(\bigvee_{i=9..12,j=5..19} (f_i S f_j)) \wedge (\bigvee_{i=1}^{12} f_i \bigvee_{i=16}^{19} f_i)))). \tag{4'}$$

The satisfiability analysis of such formulas is mush more difficult to perform due to a lot of past time operators. Nevertheless carrying out this analysis for the formulas (1), (2'-4'), (5-7) we obtain the 4-state automaton presented in Fig.3. This automaton is equivalent to the automaton presented in Fig.2 with the difference that the state XX of the automaton in Fig.2. was split into three separate states 01, 10 and

11. One can observe that these three states can be reduced to a single state XX.

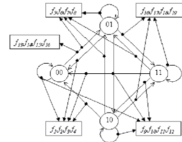

Fig.3. The 4-state automaton of the synchronous bus arbiter element.

3.6 Verifying the equivalence of the automata

The automata presented in Fig.2. and Fig.3. are equivalent. Their equivalence follows from the equivalence of the temporal logic specifications given by (1-7) and the specifications given by (1), (2'-4'), (5-7). To prove that both automata are equivalent the SMV tool has been used. In the appendix the SMV code of the 4-state automaton shown in Fig.3. is given. To prove that the automata implement correctly the synchronous bus arbiter element the same properties of the circuit are proved using the SMV code of the 4-state automaton and the SMV code of the synchronous bus arbiter element according to K.L.McMillan example syncarb.smv [4]. A slightly modified version of this example is given in the appendix.

4 Conclusions

The method presented in this paper can be used in the design of finite state automata of sequential systems as well as in the correctness analysis (verification) of the functioning of sequential systems. The method is based on the satisfiability analysis algorithm for past time linear temporal logic specifications. The algorithm was proposed to determine the satisfiability of temporal logic formulas with past time temporal operators for specific histories. A tableau-like procedure of this algorithm has been designed. To specify the histories of large systems the notion of the temporal equivalence class has been introduced. Temporal equivalence classes are very useful when dealing with input/output vectors of sequential systems. The automata of the synchronous bus arbiter circuit element have been designed. Two equivalent automata have been obtained which show the meaning of the composition and decomposition operations on temporal equivalent classes. The SMV tools has been used to verify the automata and to show that the automata implement correctly the synchronous bus arbiter circuit element.

References

[1] Z. Manna, P. Wolper, Synthesis of Communicating processes from temporal logic specifications, ACM TOPLAS, 6, 1984, pp. 68-93.

[2] A. Ursu, V. Dubenetsky, G. Gruita. Design of the Real Time Systems using Temporal Logic Specifications: A Case Study. Computer Science Journal of Moldova, 1996, Vol.4, No.1(10), pp.88-114.

[3] A. Ursu, G. Gruita and S. Zaporojan. Design and verification of the sequential systems automata using temporal logic specifications, European Design and Test Conference (EDTC'97), Paris, France, 17 - 20 March 1997, (1 page abstract).

[4] K.L. McMillan, Symbolic model checking. An approach to the state explosion problem, Doctoral thesis in the field of Computer Science, Carnegie Mellon University, may 1992.

5 Appendix

```
-- The arbiter element automaton.
-- SMV code of the automaton is written by Ursu Anatol,
-- Chisinau, August 1997.

MODULE main

VAR
  state: {s00, s01, s10, s11};
  IO_vector:
   {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,
    f16,f17,f18,f19};

ASSIGN
  init(state):=s00;
  init(IO_vector):={f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,
                    f12,f13,f14,f15,f16};
  next(state):=
   case
     (state=s00) & (IO_vector in
                      {f1,f2,f3,f4,f9,f10,f11,f12}): s00;
     (state=s00) & (IO_vector in
                      {f5,f6,f7,f8,f13,f14,f15,f16}): s01;

     (state=s01) & (IO_vector in {f5,f6,f7,f8}): s01;
     (state=s01) & (IO_vector in {f9,f10,f11,f12}): s10;
     (state=s01) & (IO_vector in {f1,f2,f3,f4}): s00;
     (state=s01) & (IO_vector in {f16,f17,f18,f19}): s11;

     (state=s11) & (IO_vector in {f5,f6,f7,f8}): s01;
     (state=s11) & (IO_vector in {f9,f10,f11,f12}): s10;
     (state=s11) & (IO_vector in {f1,f2,f3,f4}): s00;
     (state=s11) & (IO_vector in {f16,f17,f18,f19}): s11;
```

```
    (state=s10) & (IO_vector in {f5,f6,f7,f8}): s01;
    (state=s10) & (IO_vector in {f9,f10,f11,f12}): s10;
    (state=s10) & (IO_vector in {f1,f2,f3,f4}): s00;
    (state=s10) & (IO_vector in {f16,f17,f18,f19}): s11;

  1: state;
 esac;

 next(IO_vector):=
case
 (state=s00) & (IO_vector in
               {f1,f2,f3,f4,f9,f10,f11,f12}):
{f1,f2,f3,f4,f5,f6,f7,f8,f13,f14,f15,f16};
 (state=s00) & (IO_vector in
               {f5,f6,f7,f8,f13,f14,f15,f16}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};

    (state=s01) & (IO_vector in {f5,f6,f7,f8}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
    (state=s01) & (IO_vector in {f9,f10,f11,f12}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
    (state=s01) & (IO_vector in {f1,f2,f3,f4}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16};
    (state=s01) & (IO_vector in {f16,f17,f18,f19}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};

    (state=s11) & (IO_vector in {f5,f6,f7,f8}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
    (state=s11) & (IO_vector in {f9,f10,f11,f12}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
    (state=s11) & (IO_vector in {f1,f2,f3,f4}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16};
    (state=s11) & (IO_vector in {f16,f17,f18,f19}):
{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
```

```
     (state=s10) & (IO_vector in {f5,f6,f7,f8}):
 {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
     (state=s10) & (IO_vector in {f9,f10,f11,f12}):
 {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};
     (state=s10) & (IO_vector in {f1,f2,f3,f4}):
 {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16};
     (state=s10) & (IO_vector in {f16,f17,f18,f19}):
 {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f16,f17,f18,f19};

   1: IO_vector;
  esac;
DEFINE
  Request:=(IO_vector in {f9,f10,f11,f12,f13,f14,f15,f16,
                         f17,f18,f19});
  Ack:= (IO_vector in {f10,f12,f14,f16,f17,f18,f19});
  Token_in:=(IO_vector in {f5,f6,f7,f8,f13,f14,f15,f16,
                         f17,f18,f19});
  Override_in:=(IO_vector in {f3,f4,f7,f8,f11,f12,f15,
                         f16,f19});
  Grant_in:=(IO_vector in {f2,f4,f6,f8,f10,f12,f14,f16,
                         f18});
  Token_out:=(IO_vector in {f5,f6,f7,f8,f13,f14,f15,f16,
                         f17,f18,f19});
  Override_out:=(IO_vector in {f3,f4,f7,f8,f11,f12,f15,
                         f16,f17,f18,f19});
  Grant_out:=(IO_vector in {f2,f4,f6,f8});


SPEC
--   AG ((Ack -> Request) & AF (!Request | Ack))
     AG (Override_in -> Override_out) &
     !AG (Override_out -> Override_in) &
     AG (Grant_in & Request -> Ack) &
     AG (Ack -> Request) &
     AG (Token_in = Token_out) &
```

```
     AG (Request -> EF Ack) &
     !AG (Request -> Ack) &
     !AG !EF (Request -> Ack) &
     AG (Grant_out -> Grant_in) &
     AG (Grant_out = (Grant_in & !Request))

-- The arbiter element circuit.
-- The SMV code implemented according to McMillan example.
-- syncarb.smv:

MODULE main
VAR
  Persistent : boolean;
  Token : boolean;
  Request : boolean;
  Override_in: boolean;
  Grant_in: boolean;
  Token_in : boolean;

ASSIGN
  init(Request):={0,1};
  next(Request):={0,1};
  init(Override_in):={0,1};
  next(Override_in):={0,1};
  init(Grant_in):={0,1};
  next(Grant_in):={0,1};
  init(Token_in) := {0,1};
  next(Token_in) := {0,1};
  init(Token) := {0,1};
  next(Token) := Token_in;
  init(Persistent) := 0;
  next(Persistent) := Request & (Persistent | Token);

DEFINE
  Override_out := Override_in | (Persistent & Token);
```

```
  Grant_out := !Request & Grant_in;
  Ack := Request & (Persistent & Token | Grant_in);
  Token_out:= Token_in;

SPEC
--   AG ((Ack -> Request) & AF (!Request | Ack))
     AG (Override_in -> Override_out) &
     !AG (Override_out -> Override_in) &
     AG (Grant_in & Request -> Ack) &
     AG (Ack -> Request) &
     AG (Token_in = Token_out) &
     AG (Request -> EF Ack) &
     !AG (Request -> Ack) &
     !AG !EF (Request -> Ack) &
     AG (Grant_out -> Grant_in) &
     AG (Grant_out = (Grant_in & !Request))
```

A.Ursu, G.Gruita, S.Zaporojan, Received 26 October, 1997

Information Technology Department,
Technical University of Moldova,
Stefan cel Mare 168,
2012, Chisinau, Republic of Moldova,
phone: (+373-2) 497018, 497014;
fax: (+373-2) 247114
E-mail: ursu@serv2.utm.md, aursu@econ.moldnet.md
http://serv2.utm.md/∼ ursu

S.Zaporojan:
Computer Science Department,
Technical University of Moldova,
Stefan cel Mare 168,
2012, Chisinau, Republic of Moldova,
phone: (+373-2) 497016;
fax: (+373-2) 247114