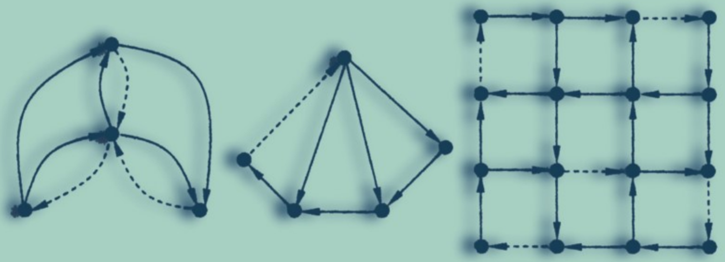
DESIGN OF EMBEDDED CONTROL SYSTEMS



Edited by

Marian Andrzej Adamski

Andrei Karatkevich

Marek Wegrzyn

Design of Embedded Control Systems

Design of Embedded Control Systems

Marian Andrzej Adamski
Andrei Karatkevich
and
Marek Wegrzyn
University of Zielona Gora, Poland

Springer

Library of Congress Cataloging-in-Publication Data

Design of embedded control systems / Marian Andrzej Adamski, Andrei Karatkevich,
Marek Wegrzyn [editors].
p. cm.
Includes bibliographical references and index.
ISBN 0-387-23630-9
1. Digital control systems—Design and construction. 2. Embedded computer systems—Design and construction. I. Adamski, M. (Marian) II. Karatkevich, Andrei. III. Wegrzyn, M. (Marek)

TJ223.M53D47 2005
629.8—dc22

2004062635

ISBN-10: 0-387-23630-9 Printed on acid-free paper.
ISBN-13: 978-0387-23630-8

Printed in the United States of America (TB/IBT)

9 8 7 6 5 4 3 2 1

springeronline.com

About the Editors

Marian Andrzej Adamski received an M.Sc. degree in electrical engineering (specialty of control engineering) from Poznan Technical University, Poland, in 1970; a Ph.D. degree in control and computer engineering from Silesian Technical University, Gliwice, Poland, in 1976; and a D.Sc. in computer engineering from Warsaw University of Technology, Poland, in 1991.

After completing his M.Sc. in 1970, he joined the research laboratory in Nuclear Electronics Company in Poznan. In 1973 he became a senior lecturer at the Technical University of Zielona Góra, Poland. From 1976 to 1991 he was employed as an assistant professor, and later from 1991 to 1992 as an associate professor. From 1993 to 1996 he was a visiting professor at University of Minho, in Braga and Guimaraes, Portugal. Currently he is a full-tenured professor of computer engineering at University of Zielona Góra, Poland. He is a chair of Computer Engineering and Electronics Institute at University of Zielona Góra.

Prof. Adamski's research includes mathematical logic and Petri nets in digital systems design, formal development of logic controller programs, and VHDL, FPLD, and FPGA in industrial applications.

Prof. M. Adamski is an author of more than 160 publications, including six books, and he holds five patents. He is a member of several international and national societies, including Committees of Polish Academy of Sciences, Polish Computer Science Society, Association for Computing Machinery (ACM), and The Institute of Electrical and Electronics Engineers (IEEE). He has supervised more than 100 M.Sc. theses and several Ph.D. dissertations. He has been a principal investigator for government-sponsored research projects and a consultant to industry. He is a member of the editorial board of *International Journal* of *Applied Mathematics and Computer Science* and a referee of international conferences and journals. He has been involved as a program and organizing committee member of several international workshops and conferences. He obtained the Scientific Award from Ministry of Higher Education and won several times the University Distinguished Teaching and Research awards.

Andrei Karatkevich received a master's degree in system engineering (1993) from Minsk Radioengineering Institute (Belarus) and Ph.D. (1998) from Belarusian State University of Informatics and Radioelectronics (Minsk). From 1998 to 2000 he was employed at this university as a lecturer. Since 1999 he has been working at University of Zielona Góra (Poland) as an Assistant Professor. Dr. Karatkevich teaches a variety of classes in computer science and computer engineering. His research interest includes digital design, theory of logical control. Petri nets, analysis and verification of concurrent algorithms, discrete systems and graph theory. He has published 40+ technical papers and several research presentations.

Marek Wegrzyn received an M.Sc. in electrical engineering (summa cum laude) from the Technical University of Zielona Góra, Poland, in 1991. Since 1991 he has been a lecturer of digital systems in Computer Engineering and Electronics Department, Faculty of Electrical Engineering at the university. He spent one academic year (1992–93) at University of Manchester Institute of Science and Technology (UMIST), Manchester, UK, working on VLSI design and HDLs (Verilog and VHDL). He has been a visiting research fellow in the Department of Industrial Electronics, University of Minho, Braga and Guimaraes, Portugal (in 1996). He received his Ph.D. in computer engineering from the Faculty of Electronics and Information Techniques at Warsaw University of Technology, Poland, in 1999. Currently, Dr. Marek Wegrzyn is an assistant professor and head of Computer Engineering Division at University of Zielona Góra, Poland.

His research interests focus on hardware description languages, Petri nets, concurrent controller designs, and information technology. His recent work includes design of dedicated FPGA-based digital systems and tools for the automatic synthesis of programmable logic. He is a referee of international conferences and journals.

Dr. Marek Wegrzyn was the 1991 recipient of the Best Young Electrical Engineer award from District Branch of Electrical Engineering Society. As the best student he obtained in 1989 a gold medal (maxima cum laude) from the rector-head of the Technical University of Zielona Góra, a Primus Inter Pares diploma, and the Nicolaus Copernicus Award from the National Student Association. He won the National Price from Ministry of Education for the distinguished Ph.D. dissertation (2000). He obtained several awards from the rector-head of the University of Zielona Góra. He has published more than 70 papers in conferences and journals. He was a coeditor of two postconference proceedings.

Foreword

A set of original results in the field of high-level design of logical control devices and systems is presented in this book. These concern different aspects of such important and long-term design problems, including the following, which seem to be the main ones.

First, the behavior of a device under design must be described properly, and some adequate formal language should be chosen for that. Second, effective algorithms should be used for checking the prepared description for correctness, for its syntactic and semantic verification at the initial behavior level. Third, the problem of logic circuit implementation must be solved using some concrete technological base; efficient methods of logic synthesis, test, and verification should be developed for that. Fourth, the task of the communication between the control device and controlled objects (and maybe between different control devices) waits for its solution. All these problems are hard enough and cannot be successfully solved without efficient methods and algorithms oriented toward computer implementation. Some of these are described in this book.

The languages used for behavior description have been descended usually from two well-known abstract models which became classic: Petri nets and finite state machines (FSMs). Anyhow, more detailed versions are developed and described in the book, which enable to give more complete information concerning specific qualities of the regarded systems. For example, the model of parallel automaton is presented, which unlike the conventional finite automaton can be placed simultaneously into several places, called *partial*. As a base for circuit implementation of control algorithms, FPGA is accepted in majority of cases.

Hierarchical Petri nets have been investigated by Andrzejewski and Miczulski, who prove their applicability to design of control devices in practical situations. Using Petri nets for design and verification of control paths is suggested by Schober, Reinsch, and Erhard, and also by Węgrzyn and Węgrzyn. A new approach to modeling and analyzing embedded hybrid control systems, based on using hybrid Petri nets and time-interval Petri nets, is proposed by

Hummel and Fengler. A memory-saving method of checking Petri nets for deadlocks and other qualities is developed by Karatkevich. A special class of reactive Petri nets with macronodes is introduced and thoroughly investigated (Gomes, Barros, and Costa). Using Petri nets for reactive system design was worked out by Adamski.

The model of sequent automaton was suggested by Zakrevskij for description of systems with many binary variables. It consists of so-called sequents—expressions defining "cause-effect" relations between events in Boolean space of input, output, and inner variables. A new method for encoding inner FSM states, oriented toward FSM decomposition, is described (Kubátová). Several algorithms were developed for assignment of partial states of parallel automata: for using in the case of synchronous automata (Pottosin) and for the asynchronous case, when race-free encoding is needed (Cheremisinova). A new technique of state exploration of statecharts specifying the behavior of controllers is suggested by Łabiak. A wide variety of formal languages is used in the object-oriented real-time techniques method, the goal of which is the specification of distributed real-time systems (Lopes, Silva, Tavares, and Monteiro).

The problem of functional decomposition is touched by Bibilo and Kirienko, who regarded it as the task of decomposing a big PLA into a set of smaller ones, and by Rawski, Łuba, Jachna, and Tomaszewicz, who applied it to circuit implementation in CPLD/FPGA architecture.

Some other problems concerning the architecture of control systems are also discussed. Architectural Description Language for using in design of embedded processors is presented by Tavares, Silva, Lima, Metrolho, and Couto. The influence of FPGA architectures on implementation of Petri net specifications is investigated by Soto and Pereira. Communication architectures of multiprocessor systems are regarded by Dvorak, who suggest some tools for their improving. A two-processor (bit-byte) architecture of a CPU with optimized interaction is suggested by Chmiel and Hrynkiewicz.

An example of application of formal design methods with estimation of their effectiveness is described by Caban, who synthesized positional digital image filters from VHDL descriptions, using field programmable devices. In another example, a technology of development and productization of virtual electronic components, both in FPGA and ASIC architectures, is presented (Sakowski, Bandzerewicz, Pyka, and Wrona).

A. Zakrevskij

Contents

Section I

Specification of Concurrent Embedded Control Systems

Chapter 1

USING SEQUENTS FOR DESCRIPTION OF CONCURRENT DIGITAL SYSTEMS BEHAVIOR

Arkadij Zakrevskij
United Institute of Informatics Problems of the National Academy of Sciences of Belarus, Surganov Str. 6, 220012, Minsk, Belarus; e-mail: zakr@newman.bas-net.by

Abstract: A model of sequent automaton is proposed for description of digital systems behavior. It consists of sequents – expressions defining "cause-effect" relations between events in the space of Boolean variables: input, output, and inner. The rules of its equivalence transformations are formulated, leading to several canonical forms. Simple sequent automaton is introduced using simple events described by conjunctive terms. It is represented in matrix form, which is intended for easing programmable logic array (PLA) implementation of the automaton. The problem of automata correctness is discussed and reduced to checking automata for consistency, irredundancy, and persistency.

Key words: logical control; behavior level; simple event; sequent automaton; PLA implementation; concurrency; correctness.

1. INTRODUCTION

Development of modern technology results in the appearance of complex engineering systems, consisting of many digital units working in parallel and often in the asynchronous way. In many cases they exchange information by means of binary signals represented by Boolean variables, and logical control devices (LCDs) are used to maintain a proper interaction between them. Design of such a device begins with defining a desirable behavior of the considered system and formulating a corresponding logical control algorithm (LCA) that must be implemented by the control device. The well-known Petri net formalism is rather often used for this purpose.

But it would be worth noting that the main theoretical results of the theory of Petri nets were obtained for pure Petri nets presenting nothing more than sets of several ordered pairs of some finite set, interpreted in a special way. To use a Petri net for LCA representation, some logical conditions and operations should be added. That is why various extensions of Petri nets have been proposed. Their common feature is that some logical variables are assigned to elements of the Petri net structure: places, transitions, arcs, and even tokens. This makes possible to represent by extended Petri nets rather complicated LCAs, but at the cost of losing the vital theoretical maintenance.

These considerations motivated developing a new approach to LCA representation[11], where Petri nets were applied together with cause-effect relations between simple discrete events (presented by elementary conjunctions). In that approach only the simplest kind of Petri nets is regarded, where arithmetic operations (used for counting the current number of tokens in a place) are changed by set operations, more convenient when solving logical problems of control algorithms verification and implementation.

According to this approach, the special language PRALU was proposed for LCA representation and used as the input language in an experimental system of CAD of LCDs[12]. A fully automated technology of LCD design was suggested, beginning with representation of some LCA in PRALU and using an intermediate formal model called sequent automaton[3–8]. A brief review of this model is given below.

2. EVENTS IN BOOLEAN SPACE

Two sets of Boolean variables constitute the interface between an LCD and some object of control: the set X of *condition variables* $x_1, \dots, x_n$ that present some information obtained from the object (delivered by some sensors, for example) and the set Y of *control variables* $y_1, \dots, y_m$ that present control signals sent to the object. Note that these two sets may intersect – the same variable could be presented in both sets when it is used in a feedback. From the LCDs point of view X may be considered as the set of input variables, and Y as the set of output variables. In case of an LCD with memory the third set Z is added interpreted as the set of inner variables. Union of all these sets constitutes the general set W of Boolean variables.

$2^{|W|}$ different combinations of values of variables from W constitute the Boolean space over W($|W|$ denotes the cardinality of set W). This Boolean space is designated below as BS(W). Each of its elements may be regarded as a global state of the system, or as the corresponding event that occurs when the system enters that state. Let us call such an event *elementary*. In the same way, the elements of Boolean spaces over X, Y, and Z may be regarded as input states, output states, and inner states, as well as corresponding events.

Besides these, many more events of other types may be taken into consideration. Generally, every subset of BS(W) may be interpreted as an event that occurs when some element from BS(W) is realized; i.e., when the variables from W possess the corresponding combination of values. In this general case the event is called *complicated* and could be presented by the characteristic Boolean function of the regarded subset. Therefore, the number of complicated events coincides with the number of arbitrary Boolean functions of $|W|$ variables.

From the practical point of view, the following two types of events deserve special consideration: basic events and simple events.

Basic events are represented by literals – symbols of variables or their negations – and occur when these variables take on corresponding values. For example, basic event a occurs when variable a equals 1, and event c' occurs when $c = 0$. The number of different basic events is $2|W|$.

Simple events are represented by elementary conjunctions and occur when these conjunctions take value 1. For example, event $ab'f$ occurs when $a = 1$, $b = 0$, and $f = 1$. The number of different simple events is $3^{|W|}$, including the trivial event, when values of all variables are arbitrary.

Evidently, the class of simple events absorbs elementary events and basic events. Therefore, elementary conjunction k_i is the general form for representation of events i of all three introduced types; it contains symbols of all variables in the case of an elementary event and only one symbol when a basic event is regarded. One event i can realize another event j – it means that the latter always comes when the former comes. It follows from the definitions that it occurs when conjunction k_i implicates conjunction k_j; in other words, when k_j can be obtained from k_i by deleting some of its letters. For example, event $abc'de'$ realizes events $ac'd$ and $bc'e'$, event $ac'd$ realizes basic events a, c', and d, and so on. Hence, several different events can occur simultaneously, if only they are not orthogonal.

3. SEQUENT AUTOMATON

The behavior of a digital system is defined by the rules of changing its state. A standard form for describing such rules was suggested by the well-developed classical theory of finite automata considering relations between the sets of input, inner, and output states. Unfortunately, this model becomes inapplicable for digital systems with many Boolean variables – hundreds and more. That is why a new formal model called *sequent automaton* was proposed[3–5]. It takes into account the fact that interaction between variables from W takes place within comparatively small groups and has functional character, and it suggests means for describing both the control unit of the system and the object of control – the body of the system.

Sequent automaton is a logical dynamic model defined formally as a system S of *sequents* s_i. Each sequent s_i has the form $f_i \mid\!- k_i$ and defines the cause-effect relation between a complicated event represented by Boolean function f_i and a simple event k_i represented by conjunction term k_i; $\mid\!-$ is the symbol of the considered relation. Suppose that function f_i is given in disjunctive normal form (DNF).

The expression $f_i \mid\!- k_i$ is interpreted as follows: if at some moment function f_i takes value 1, then immediately after that k_i must also become equal to 1 – by this the values of all variables in k_i are defined uniquely. In such a way a separate sequent can present a definite demand to the behavior of the discrete system; and the set S as a whole, the totality of such demands.

Note that the variables from X may appear only in f_i and can carry information obtained from some sensors; the variables from Y present control signals and appear only in k_i; and the variables from Z are feedback variables that can appear both in f_i and k_i.

The explication of "immediately after that" depends greatly on the accepted time model. It is different for two kinds of behavior interpretation, which could be used for sequent automata, both of practical interest: synchronous and asynchronous.

We shall interpret system S mostly as a *synchronous* sequent automaton. In this case the behavior of the automaton is regarded in discrete time t, the sequence of moments $t_0, t_1, t_2, \dots, t_l, t_{l+1}, \dots$. At a current transition from t_l to t_{l+1} all such sequents s_i for which $f_i = 1$ are executed simultaneously, and as a result all corresponding conjunctions k_i turn to 1 (all their factors take value 1). In that case "immediately after that" means "at the next moment."

Suppose that if some of the inner and output variables are absent in conjunctions k_i of executed sequents, they preserve their previous values. That is why the regarded sequent automata are called *inertial*[9]. Hence a new state of the sequent automaton (the set of values of inner variables), as well as new values of output variables, is defined uniquely.

Sometimes the initial state of the automaton is fixed (for moment t_0); then the automaton is called *initial*. The initial state uniquely determines the set R of all reachable states. When computing it, it is supposed that all input variables are free; i.e., by any moment t_l they could take arbitrary combinations of values. Let us represent set R by characteristic Boolean function φ of inner variables, which takes value 1 on the elements from R. In the case of noninitialized automata it is reasonable to consider that $\varphi = 1$.

Under *asynchronous* interpretation the behavior of sequent automaton is regarded in continuous time. There appear many more hard problems of their analysis connected with races between variables presented in terms k_i, especially when providing the automaton with the important quality of correctness.

4. EQUIVALENCE TRANSFORMATIONS AND CANONICAL FORMS

Let us say that sequent s_i is *satisfied* in some engineering system if event f_i is always followed by event k_i and that sequent s_i *realizes* sequent s_j if the latter is satisfied automatically when the former is satisfied.

Affirmation 1. Sequent s_i realizes sequent s_j if and only if $f_j \Rightarrow f_i$ and $k_i \Rightarrow k_j$, where $\Rightarrow$ is the symbol of formal implication.

For instance, sequent $ab \vee c \mid\!- uv'$ realizes sequent $abc \mid\!- u$. Indeed, $abc \Rightarrow ab \vee c$ and $uv' \Rightarrow u$.

If two sequents s_i and s_j realize each other, they are *equivalent*. In that case $f_i = f_j$ and $k_i = k_j$.

The relations of realization and equivalence can be extended onto sequent automata S and T. If S includes in some form all demands contained in T, then S realizes T. If two automata realize each other, they are equivalent.

These relations are easily defined for *elementary sequent automata* S^e and T^e, which consist of *elementary sequents*. The left part of such a sequent presents an elementary event in BS($X \cup Z$), and the right part presents a basic event (for example, $ab'cde' \mid\!- q$, where it is supposed that $X \cup Z = \{a, b, c, d, e\}$). S^e realizes T^e if it contains all sequents contained in T^e. S^e and T^e are equivalent if they contain the same sequents. It follows from this that the elementary sequent automaton is a *canonical form*.

There exist two basic equivalencies formulated as follows.

Affirmation 2. Sequent $f_i \vee f_j \mid\!- k$ is equivalent to the pair of sequents $f_i \mid\!- k$ and $f_j \mid\!- k$.

Affirmation 3. Sequent $f \mid\!- k_i k_j$ is equivalent to the pair of sequents $f \mid\!- k_i$ and $f \mid\!- k_j$.

According to these affirmations, any sequent can be decomposed into a series of elementary sequents (which cannot be decomposed further). This transformation enables to compare any sequent automata, checking them for binary relations of realization and equivalence. Affirmations 2 and 3 can be used for equivalence transformations of sequent automata by elementary operations of two kinds: splitting sequents (replacing one sequent by a pair) and merging sequents (replacing a pair of sequents by one, if possible).

Elementary sequent automaton is useful for theoretical constructions but could turn out quite noneconomical when regarding some real control systems. Therefore two more canonical forms are introduced.

The *point sequent automaton* S^{p} consists of sequents in which all left parts represent elementary events (in BS($X \cup Z$)) and are different. The corresponding right parts show the responses. This form can be obtained from elementary sequent automaton S^e by merging sequents with equal left parts.

The *functional sequent automaton* S^f consists of sequents in which all right parts represent basic events in $\mathrm{BS}(Z \cup Y)$ and are different. So the sequents have the form $f_i^1 \mid\!- u_i$ or $f_i^0 \mid\!- u_i'$, where variables $u_i \in Z \cup Y$, and the corresponding left parts are interpreted as switching functions for them: ON functions f_i^1 and OFF functions f_i^0. S^f can be obtained from S^e by merging sequents with equal right parts.

Note that both forms S^p and S^f can also be obtained from arbitrary sequent automata by disjunctive decomposition of the left parts of the sequents (for the point sequent automaton) or conjunctive decomposition of the right parts (for the functional one).

5. SIMPLE SEQUENT AUTOMATON

Now consider a special important type of sequent automata, a *simple sequent automaton*. It is defined formally as a system S of *simple sequents*, expressions $k_i' \mid\!- k_i''$ where both k_i' and k_i'' are elementary conjunctions representing simple events. This form has a convenient matrix representation, inasmuch as every elementary conjunction can be presented as a ternary vector.

Let us represent any simple sequent automaton by two ternary matrices: a *cause matrix* $\boldsymbol{A}$ and an *effect matrix* $\boldsymbol{B}$. They have equal number of rows indicating simple sequents, and their columns correspond to Boolean variables – input, output, and inner.

Example. Two ternary matrices

$$\boldsymbol{A} = \begin{array}{c} \begin{array}{cccccc} a & b & c & p & q & r \end{array} \\ \begin{pmatrix} 1 & - & - & - & 0 & - \\ - & 0 & 1 & 1 & - & - \\ 0 & 1 & - & - & 1 & 1 \\ - & - & 0 & - & - & 0 \\ - & - & 0 & 1 & 0 & - \end{pmatrix} \end{array}, \qquad \boldsymbol{B} = \begin{array}{c} \begin{array}{ccccccc} p & q & r & u & v & w & z \end{array} \\ \begin{pmatrix} - & 1 & - & - & 1 & - & 1 \\ - & - & 0 & 1 & - & 0 & - \\ 1 & 0 & - & - & 1 & - & 0 \\ 0 & - & - & - & - & 1 & - \\ - & 1 & 1 & 0 & - & 1 & - \end{pmatrix} \end{array}$$

represent the following system of simple sequents regarded as a simple sequent automaton:

$$aq' \mid\!- qvz,$$

$$b'cp \mid\!- r'uw',$$

$$a'bqr \mid\!- pq'vz',$$

$$c'r' \mid\!- p'w,$$

$$c'pq' \mid\!- qru'w.$$

Here $X = \{a, b, c\}, Y = \{u, v, w, z\}, Z = \{p, q, r\}$.

It has been noted[1] that, to a certain extent, simple sequents resemble the sequents of the theory of logical inference introduced by Gentzen[2]. The latter are defined as expressions

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m,$$

which connect arbitrary logic formulae $A_1, \ldots, A_n, B_1, \ldots, B_m$ and are interpreted as implications

$$A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m.$$

The main difference is that any simple sequent $k_i' |\!- k_i''$ presents not a pure logical but a cause-effect relation: event k_i'' is generated by event k_i' and appears after it, so we cannot mix variables from k_i' with variables from k_i''.

But sometimes we may discard this time aspect and consider terms k_i' and k_i'' on the same level; for instance, when looking for stable states of the regarded system. In that case, sequent $k_i' |\!- k_i''$ could be formally changed for implication $k_i' \rightarrow k_i''$ and subjected further to Boolean transformations, leading to equivalent sets of Gentzen sequents and corresponding sets of standard disjuncts usual for the theory of logical inference.

For example, the system of simple sequents

$$ab |\!- cd', a'b' |\!- cd, a'b |\!- c$$

may be transformed into the following system of disjuncts

$$a' \vee b' \vee c, a' \vee b' \vee d', a \vee b \vee c, a \vee b \vee d, a \vee b' \vee c.$$

6. APPLICATION IN LOGIC DESIGN

The model of simple sequent automaton is rather close to the well-known technique of disjunctive normal forms (DNFs) used for hardware implementation of systems of Boolean functions. Indeed, each row of matrix $\boldsymbol{A}$ may be regarded as a conjunctive term (product), and each column in $\boldsymbol{B}$ defines DNFs for two switching functions of the corresponding output or inner variable: 1's indicate terms entering ON functions, while 0's indicate terms which enter OFF functions. Note that these DNFs can be easily obtained by transforming the regarded automaton into S^f-form and then changing expressions $f_i^1 |\!- u_i$ for $u_i^1 = f_i^1$ and $f_i^0 |\!- u_i'$ for $u_i^0 = f_i^0$. For the same example

$$p^1 = a'bqr, p^0 = c'r'; q^1 = aq' \vee c'pq', q^0 = a'bqr; r^1 = c'pq', r^0 = b'cp;$$

$$u^1 = b'cp, u^0 = c'pq'; v^1 = aq' \vee a'bqr; w^1 = c'r' \vee c'pq', w^0 = b'cp;$$

$$z^1 = aq', z^0 = a'bqr.$$

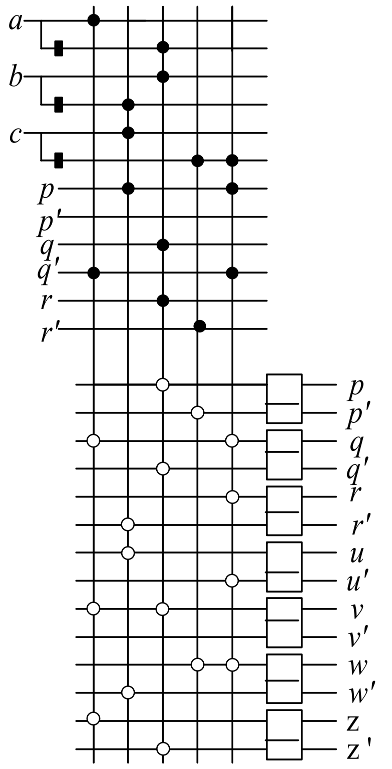


Figure 1-1. PLA implementation of a simple sequent automaton.

It is seen from here that the problem of constructing a simple sequent automaton with minimum number of rows is similar to that of the minimization of a system of Boolean functions in the class of DNFs known as a hard combinatorial problem. An approach to its solving was suggested in Refs. 7 and 8.

The considered model turned out to be especially convenient for representation of programmable logic arrays (PLAs) with memory on RS-flip-flops. It is also used in methods of automaton implementation of parallel algorithms for logical control described by expressions in PRALU[11].

Consider a simple sequent automaton shown in the above example. It is implemented by a PLA represented in Fig. 1-1. It has three inputs (a, b, c) supplied with inverters (NOT elements) and four outputs (u, v, w, z) supplied with RS flip-flops. So its input and output lines are doubled. The six input lines are intersecting with five inner ones, and at some points of intersection transistors are placed. Their disposition can be presented by a Boolean matrix easily obtained from matrix $\boldsymbol{A}$ and determines the AND plane of the PLA. In a similar way the OR plane of the PLA is found from matrix $\boldsymbol{B}$ and realized on the intersection of inner lines with 14 output lines.

7. CHECKING FOR CORRECTNESS

In general, correctness is a quality of objects of some type, defined as the sum of several properties, which are considered reasonable and necessary[10].

Let us enumerate such properties first for synchronous sequent automata. Evidently, for any sequent s_i that carries some information, inequalities $f_i \neq 0$ and $k_i \neq 1$ should hold, to avoid trivial sequents.

Sequents s_i and s_j are called *parallel* if they could be executed simultaneously. A necessary and sufficient condition of parallelism for a noninitialized automaton is relation $f_i \wedge f_j \neq 0$ for the initialized relation $f_i \wedge f_j \wedge \varphi \neq 0$.

First of all, any sequent automaton should be *consistent*; that is very important. This means that for any parallel sequents s_i and s_j, relation $k_i \wedge k_j \neq 0$ must hold. Evidently, this condition is necessary, inasmuch as by its violation some variable exists that must take two different values, which is impossible.

The second quality is not so necessary for sequent automata as the first one, but it is useful. It is *irredundancy*. A system S is *irredundant* if it is impossible to remove from it a sequent or only a literal from a sequent without violating the functional properties of the system. For example, it should not have "nonreachable" sequents, such as s_i for which $f_i \wedge \varphi = 0$.

It is rather easy to check a simple sequent automaton for consistency. An automaton represented by ternary matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is obviously consistent if for any orthogonal rows in matrix $\boldsymbol{B}$ the corresponding rows of matrix $\boldsymbol{A}$ are also orthogonal. Note that this condition is satisfied in Example.

One more useful quality called *persistency* is very important for asynchronous sequent automata. To check them for this quality it is convenient to deal with the functional canonical form.

The point is that several sequents can be executed simultaneously and if the sequent automaton is asynchronous, these sequents (called *parallel*) could compete, and the so-called *race* could take place. The automaton is *persistent* if the execution of one of the parallel sequents does not destroy the conditions for executing other sequents.

Affirmation 4. In a persistent asynchronous sequent automaton for any pair of parallel sequents

$f_i^1 \mid\!- u_i$ and $f_j^1 \mid\!- u_j$,

$f_i^0 \mid\!- u_i'$ and $f_j^1 \mid\!- u_j$,

$f_i^1 \mid\!- u_i$ and $f_j^0 \mid\!- u_j'$,

$f_i^0 \mid\!- u_i'$ and $f_j^0 \mid\!- u_j'$, the corresponding relation

$$f_i^1 f_j^1 : u_i' u_j' \Rightarrow (f_i^1 : u_i' u_j)(f_j^1 : u_i u_j'),$$
$$f_i^0 f_j^1 : u_i u_j' \Rightarrow (f_i^0 : u_i u_j)(f_j^1 : u_i' u_j'),$$
$$f_i^1 f_j^0 : u_i' u_j \Rightarrow (f_i^1 : u_i' u_j')(f_j^0 : u_i u_j),$$
$$f_i^0 f_j^0 : u_i u_j \Rightarrow (f_i^0 : u_i u_j')(f_j^0 : u_i' u_j),$$

should hold, where expression $f : k$ means the result of substitution of those variables of function f that appear in the elementary conjunction k by the values satisfying equation $k = 1$.

The proof of this affirmation can be found in Ref. 9.

ACKNOWLEDGMENT

This research was supported by ISTC, Project B-986.

REFERENCES

1. M. Adamski, *Digital Systems Design by Means of Rigorous and Structural Method.* Wydawnictwo Wyzszej Szkoly Inzynierskiej, Zielona Gora (1990) (in Polish).
2. G. Gentzen, Untersuchungen über das Logische Schließen. *Ukrainskii Matematicheskii Zhurnal*, **39** 176–210, 405–431 (1934–35).
3. V.S. Grigoryev, A.D. Zakrevskij, V.A. Perchuk, *The Sequent Model of the Discrete Automaton. Vychislitelnaya Tekhnika v Mashinostroenii*. Institute of Engineering Cybernetics, Minsk, 147–153 (March 1972) (in Russian).
4. V.N. Zakharov, *Sequent Description of Control Automata*. Izvestiya AN SSSR, No. 2 (1972) (in Russian).
5. V.N. Zakharov, *Automata with Distributed Memory*. Energia, Moscow (1975) (in Russian).
6. A.D. Zakrevskij, V.S. Grigoryev, A system for synthesis of sequent automata in the basis of arbitrary DNFs. In: *Problems of Cybernetics. Theory of Relay Devices and Finite Automata*. VINITI, Moscow, 157–166 (1975) (in Russian).
7. A.D. Zakrevskij, *Optimizing Sequent Automata. Optimization in Digital Devices Design*. Leningrad, 42–52 (1976) (in Russian).
8. A.D. Zakrevskij, Optimizing transformations of sequent automata. *Tanul. MTA SeAKJ*, **63**, Budapest, 147–151 (1977) (in Russian).
9. A.D. Zakrevskij, *Logical Synthesis of Cascade Networks*. Nauka Moscow (1981) (in Russian).
10. A.D. Zakrevskij, The analysis of concurrent logic control algorithms. In: L. Budach, R.G. Bukharaev, O.B. Lupanov (eds.), *Fundamentals of Computation Theory*. Lecture Notes in Computer Science, Vol. 278. Springer-Verlag, Berlin Heidelberg New York London Paris Tokyo, 497–500 (1987).

11. A.D. Zakrevskij, *Parallel Algorithms for Logical Control*. Institute of Engineering Cybernetics, Minsk (1999) (in Russian).
12. A.D. Zakrevskij, Y.V. Pottosin, V.I. Romanov, I.V. Vasilkova, Experimental system of automated design of logical control devices. In: *Proceedings of the International Workshop "Discrete Optimization Methods in Scheduling and Computer-Aided Design*", Minsk pp. 216–221 (September 5–6, 2000).

Chapter 2

FORMAL LOGIC DESIGN OF REPROGRAMMABLE CONTROLLERS

Marian Adamski
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgorna 50, 65-246 Zielona Góra, Poland; e-mail: M.Adamski@iie.uz.zgora.pl

Abstract: The goal of the paper is to present a formal, rigorous approach to the design of logic controllers, which are implemented as independent control units or as central control parts inside modern reconfigurable microsystems. A discrete model of a dedicated digital system is derived from the control interpreted Petri net behavioral specification and considered as a modular concurrent state machine. After hierarchical and distributed local state encoding, an equivalent symbolic description of a sequential system is reflected in field programmable logic by means of commercial CAD tools. The desired behavior of the designed reprogrammable logic controller can be validated by simulation in a VHDL environment.

Key words: Petri nets; logic controllers; hardware description languages (HDL); field programmable logic.

1. INTRODUCTION

The paper covers some effective techniques for computer-based synthesis of reprogrammable logic controllers (RLCs), which start from the given interpreted Petri net based behavioral specification. It is shown how to implement parallel (concurrent) controllers[1,4,8,14] in field programmable logic (FPL). The symbolic specification of the Petri net is considered in terms of its local state changes, which are represented graphically by means of labeled transitions, together with their input and output places. Such simple subnets of control interpreted Petri nets are described in the form of decision rules – logic assertions in propositional logic, written in the Gentzen sequent style[1,2,12].

Formal expressions (sequents), which describe both the structure of the net and the intended behavior of a discrete system, may be verified formally in the context of mathematical logic and Petri net theory. For professional validation by simulation and effective synthesis, they are automatically transformed into intermediate VHDL programs, which are accepted by industrial CAD tools.

The main goal of the proposed design style is to continuously preserve the direct, self-evident correspondence between modular interpreted Petri nets, symbolic specification, and all considered hierarchically structured implementations of modeled digital systems, implemented in configurable or reconfigurable logic arrays.

The paper presents an extended outline of the proposed design methodology, which was previously presented in *DESDes'01* Conference Proceedings[3]. The modular approach to specification and synthesis of concurrent controllers is applied, and a direct hierarchical mapping of Petri nets into FPL is demonstrated. The author assumes that the reader has a basic knowledge of Petri nets[5,9,10,13,14]. The early basic ideas related with concurrent controller design are reported in the chapter in Ref. 1. The author's previous work on reprogrammable logic controllers has been summarized in various papers[2,4,6,8]. Several important aspects of Petri net mapping into hardware are covered in books[3,13,14]. The implementation of Petri net based controllers from VHDL descriptions can be found in Refs. 2, 6, and 13. Some arguments of using Petri nets instead of linked sequential state machines are pointed in Ref. 9.

2. CONCURRENT STATE MACHINE

In the traditional *sequential finite state machine* (SFSM) model, the logic controller changes its *global internal states*, which are usually recognized by their mnemonic names. The set of all the possible internal states is finite and fixed. Only one *current state* is able to hold (be active), and only one *next state* can be chosen during a particular *global state change*. The behavioral specification of the modeled sequential logic controller is frequently given as a state graph (diagram) and may be easily transformed into state machine–Petri net (SM-PN), in which only one current *marked place*, representing the active state, contains a *token*. In that case, the state change of controller is always represented by means of a *transfer transition*, with only one input and only one output place. The traditional single SFSM based models are useful only for the description of simple tasks, which are manually coordinated as linked state machines with a lot of effort[9]. The equivalent SFSM model of highly concurrent system is complicated and difficult to obtain, because of the state space explosion.

In the modular Petri net approach, a *concurrent finite state machine* (CFSM) simultaneously holds several *local states*, and several local state changes can occur independently and concurrently. The *global states* of the controller, included into the equivalent SFSM model, can be eventually deduced as maximal subsets of the local states, which simultaneously hold (*configurations*). They correspond to all different maximal sets of marked places, which are obtained during the complete execution of the net. They are usually presented in compact form as vertices in Petri net reachability graph[5,10,13]. It should be stressed that the explicitly obtained behaviorally equivalent *transition systems* are usually complex, both for maintenance and effective synthesis. The methodology proposed in the paper makes it possible to obtain a correctly encoded and implemented transition system directly from a Petri net, without knowing its global state set.

The novel tactic presented in this paper is based on a hierarchical decomposition of Petri nets into self-contained and structurally ordered modular subsets, which can be easily identified and recognized by their common parts of the internal state code. The total codes of the related modular Petri net subnets, which are represented graphically as *macroplaces*, can be obtained by means of a simple hierarchical superposition (merging) of appropriate codes of individual places. On the other hand, the code of a particular place includes specific parts, which precisely define all hierarchically ordered macroplaces, which contain the considered place inside. In such a way any separated part of a behavioral specification can be immediately recognized on the proper level of abstraction and easily found in the regular cell structure (logic array). It can be efficiently modified, rejected, or replaced during the validation or redesign of the digital circuit.

Boolean expressions called *predicate labels* or *guards* depict the external conditions for transitions, so they can be enabled. One of enabled transition occurs (it *fires*). Every immediate *combinational Moore type output signal y* is linked with some *sequentially related places*, and it is activated when one of these places holds a token. Immediate *combinational Mealy type output signals* are also related with proper subsets of sequentially related places, but they also depend on relevant (valid) input signals or internal signals. The Mealy type output is active if the place holds a token and the correlated logic conditional expression is true.

The implemented Petri net should be determined (without conflicts), safe, reversible, and without deadlocks[5,7]. For several practical reasons the synchronous hardware implementations of Petri nets[4,6,7] are preferred. They can be realized as dedicated digital circuits, with an internal state register and eventual output registers, which are usually synchronized by a common clock. It is considered here that all enabled concurrent transitions can fire independently in any order, but nearly immediately.

In the example under consideration (Fig. 2-1), Petri net places $P = \{p1–p9\}$ stand for the local states $\{P1–P9\}$ of the implemented logic controller. The

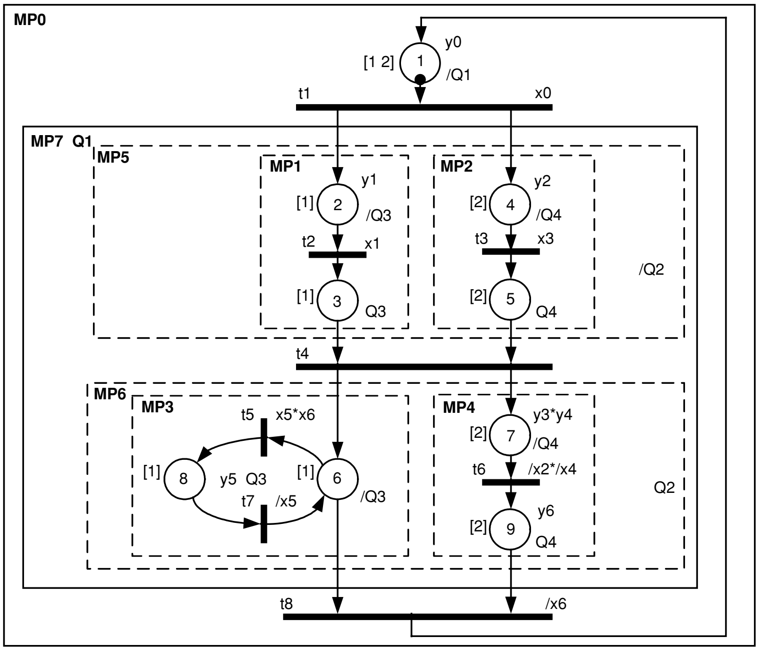


Figure 2-1. Modular, hierarchical and colored control interpreted Petri net.

Petri net transitions $T = \{t1–t8\}$ symbolize all the possible *local state changes {T1–T9}*. The Petri net places are hierarchically grouped as nested modular macroplaces *MP0-MP7*. The Petri net describes a controller with inputs $x0–x6$ and outputs $y_0–y_6$. The controller contains an internal state register with flip-flops *Q1–Q4*. The state variables structurally encode places and macroplaces to be implemented in hardware.

The direct mapping of a Petri net into field programmable logic (FPL) is based on a self-evident correspondence between a place and a clearly defined bit-subset of a state register. The place of the Petri net is assigned only to the particular part of the register block (only to selected variables from internal state register *Q1–Q4*). The beginning of local state changes is influenced by the edge of the clock signal, giving always, as a superposition of excitations, the predicted final global state in the state register. The high-active input values are denoted as *xi*, and low-active input values as */xi*.

The net could be SM-colored during the specification process, demonstrating the paths of recognized intended sequential processes (state machines subnets). These colors evidently help the designer to intuitively and formally validate the consistency of all sequential processes in the developed discrete state model[4]. The colored subnets usually replicate Petri net place invariants. The invariants of the top-level subnets can be hierarchically determined by invariants of its subnets. If a given net or subnet has not been previously colored by a designer during specification, it is possible to perform the coloring procedure by means of analysis of configurations. Any two concurrent places or macroplaces, which are marked simultaneously, cannot share the same color. It means that coloring of the net can be obtained by coloring the *concurrency graph*, applying the well-known methods taken from the graph theory[1,2]. For some classes of Petri nets, the concurrency relation can be found without the derivation of all the global state space[4,7,13]. The colors (*[1], [2]*), which paint the places in Fig. 2-1, separate two independent sequences of local state changes. They are easy to find as closed chains of transitions, in which selected input and output places are painted by means of identical colors.

The equivalent interpreted SM Petri net model, derived from equivalent transition system description (interpreted Petri net reachability graph of the logic controller), is given in Fig. 2-2.

The distribution of Petri net tokens among places, before the firing of any transition, can be regarded as the identification of the *current global state M*. Marking *M* after the firing of any enabled transition is treated as the *next global*

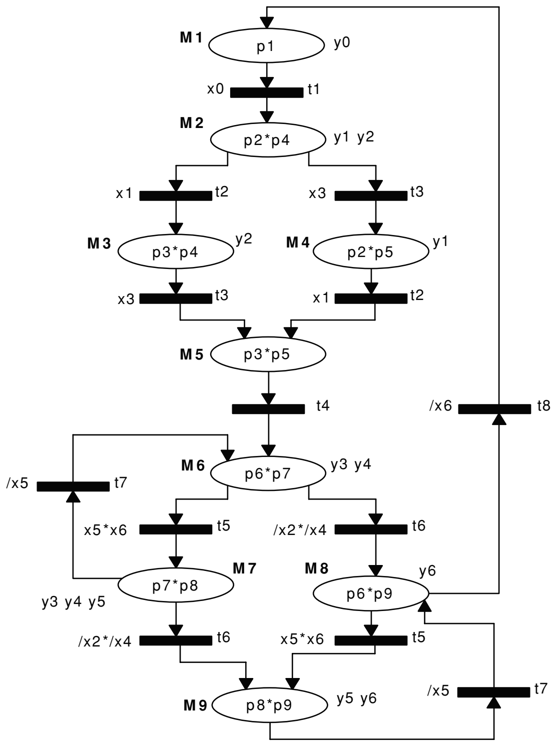


Figure 2-2. Global states of control interpreted Petri net. Transition system modeled as equivalent state machine Petri net.

state @M. From the present global internal state M, the modeled controller goes to the next internal global state @M, generating the desired combinational immediate output signals y and registered @y output signals.

There are 9 places describing global states M1–M9 and 13 transitions between 13 pairs of global states. Such an implicit formal structure really exists in hardware, although its internal structure could be unknown, because eventually deduced global state codes are immediately read from the state register as a consistent superposition of local state codes. Since a Moore-type output should be stable during the entire clock period, it can also be produced as a *registered Moore-type output @y*. The registered Moore-type output signals should be predicted before the local state changes.

3. LOGIC CONTROLLER AS ABSTRACT REASONING SYSTEM IMPLEMENTED IN DIGITAL HARDWARE

The well-structured formal specification, which is represented in the human-readable language, has a direct impact on the validation, formal verification, and implementation of digital microsystems in FPL. The declarative, logic-based specification of the Petri net can increase the efficiency of the concurrent (parallel) controller design. The proposed model of a concurrent state machine can be considered within a framework of the concept of sequent parallel automaton, developed by Zakrevskij[3,14]. Here, a control automaton, with discrete elementary states and composite super-states, is treated as a dynamic inference system, based on Gentzen sequent logic[1,2,12].

After analysis of some behavioral and structural properties of the Petri net[5,7,10,13,14], a discrete-event model is related with a knowledge-based, textual, descriptive form of representation. The syntactic and semantic compatibility between Petri net descriptions and symbolic conditional assertions are kept as close as possible. The symbolic sequents-axioms may include elements, taken from temporal logic, especially operator "next" @[11]. Statements about the discrete behavior of the designed system (behavioral axioms) are represented by means of sequents-assertions, forming the rule-base of the decision system, implemented in reconfigurable hardware. Eventual complex sequents are formally, step by step, transformed into the set of the equivalent sequent-clauses, which are very similar to elementary sequents[1,3,14]. The simple *decision rules*, which are transformed into reprogrammable hardware, can be automatically mapped into equivalent VHDL statements on RTL level[2,6,13]. The next steps of design are performed by means of professional CAD tools.

The implicit or explicit interpreted reachability graph of the net is considered here only as a conceptual supplement: compact description of an equivalent discrete transition system (Fig. 2-2), as well as Kripke interpretation structure[11] for symbolic logic conditionals.

4. STRUCTURED LOCAL STATE ENCODING

The simplest technique for Petri net place encoding is to use one-to-one mapping of places onto flip-flops in the style of one-hot state assignment. In that case, a name of the place becomes also a name of the related flip-flop. The flip-flop *Qi* is set to 1 if and only if the particular place *pi* holds the token. In such a case it is a popular mistake to think that other state variables *Qj, . . . ,Qk* have simultaneously "*don't care values*." It is evidently seen from the reachability graph (Fig. 2-2) that the places *from the same P-invariant*, which are sequentially related with the considered place *pi*, do not hold tokens, and consequently all flip-flops used for their encoding have to be set to logical 0. On the other hand, the places from the same configuration but with different colors, which are concurrently related with the selected place *pi*, have strictly defined, but not necessarily explicitly known, markings. The only way of avoiding such misunderstanding of the *concurrent one-hot encoding* is the assumption that the considered place, marked by token, can be *recognized* by its own, private flip-flop, which is set to logical 1, and that signals from other flip-flops from the state register are always fixed but usually unknown.

In general encoding, places are recognized by their particular coding conjunctions[1], which are formed from state variables, properly chosen from the fixed set of flip-flop names $\{Q1, Q2, \dots, Qk\}$. The registered output variables $\{Y\}$ can be eventually merged with the state variable set $\{Q\}$ and economically applied to the Petri net place encoding as local state variables. For simplicity, the encoded and implemented place *pi* is treated as a complex signal *Pi*.

The local states, which are simultaneously active, must have nonorthogonal codes. It means that the Boolean expression formed as a conjunction of *coding terms* for such concurrent places is always satisfied (always different from logical 0). The configuration of concurrent places gives as superposition of coding conjunctions a unique code of the considered global state.

The local states, which are not concurrent, consequently belong to at least one common sequential process (Figs. 2-2, 2-3). Their symbols are not included in the same vertex of the reachability graph, so they may have orthogonal codes.

The code of a particular place or macroplace is represented by means of a vector composed of $\{0, 1, \dots, *\}$, or it is given textually as a related Boolean term. The symbols of the values for logic signals 0, 1, and "don't care" have the usual meanings. The symbol * in the vector denotes "explicitly don't know" value (0 or 1, but no "don't care"). In expressions, the symbol / denotes the operator of logic negation, and the symbol * represents the operator of logic conjunction. An example[3] of a heuristic hierarchical local state assignment *[Q1, Q2, Q3, Q4]* is as follows:

```
P1[1,2] = 0 - - -     QP1= /Q1
P2[1]   = 1 0 0 *     QP2= Q1*/Q2*/Q3
P3[1]   = 1 0 1 *     QP3= Q1*/Q2*Q3
P4[2]   = 1 0 * 0     QP4= Q1*/Q2*/Q4
P5[2]   = 1 0 * 1     QP5= Q1*/Q2*Q4
P6[1]   = 1 1 0 *     QP6= Q1*Q2*/Q3
P7[2]   = 1 1 * 0     QP7= Q1*Q2*/Q4
P8[1]   = 1 1 1 *     QP8= Q1*Q2*Q3
P9[2]   = 1 1 * 1     QP9= Q1*Q2*Q4
```

The global state encoding is correct if all vertices of the reachability graph have different codes. The total code of the global state (a vertex of the reachability graph) can be obtained by merging the codes of the simultaneously marked places. Taking as an example some global states (vertices of the reachability graph; Fig. 2-2), we obtain

QM3 = QP3 * QP4 = Q1 */Q2 * Q3 */Q4;

QM4 = QP2 * QP5 = Q1 */Q2 */Q3 * Q4.

5. MODULAR PETRI NET

Modular and hierarchical Petri nets can provide a unified style for the design of logic controllers, from an initial behavioral system description to the possibly different hierarchical physical realizations. The concurrency relation between subnets can be partially seen from the distribution of colors. Colors[5] are attached explicitly to the places and macroplaces, and implicitly to the transitions, arcs, and tokens[2,3,4]. Before the mapping into hardware, the Petri net is hierarchically encoded. The Petri net from Fig. 2-2 can be successfully reduced to one compound multiactive macroplace *MP0*. The colored hierarchy tree in Fig. 2-3 graphically represents both the hierarchy and partial concurrency relations between subnets (modules). It contains a single ordinary monoactive place *P1[1,2]*, coded as QP1 = /Q1, and a multiactive double-macroplace *MP7[1,2]*, coded as QMP7 = Q1, which stands for other hierarchically nested subnets, from lower levels of abstraction containing places p1–p9. The macroplace *MP7[1,2]* is built of the sequentially related macroplaces *MP5[1,2]* and *MP6[1,2]*, which are coded respectively as Q1*/Q2 and Q1*Q2.

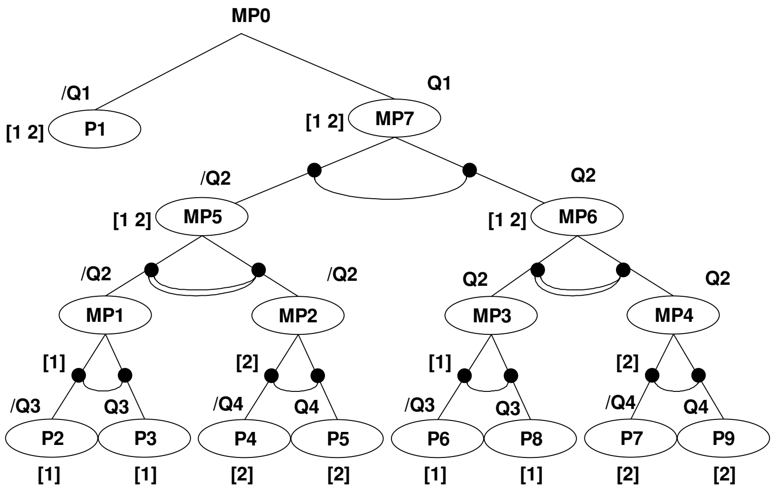


Figure 2-3. Hierarchy tree.

It should be mentioned that parallel macroplaces may obtain exactly the same top-level codes. The macroplace *MP5[1,2]* consists of two parallel macroplaces *MP1[1]* and *MP2[2]*, which are not recognized by different conjunctions inside MP5. The macroplace *MP6[1,2]* appears as an abstraction of two other parallel macroplaces *MP3[1]* and *MP4[2]*. The macroplaces *MP1[1], MP2[2], MP3[1]*, and *MP4[2]* are directly extracted from the initial Petri net as elementary sequential subnets.

The concurrency relation between subnets, which belong to the same macroplace, can be expressed graphically by means of additional double or single lines (Fig. 2-3). The code of the macroplace or place on a lower level of hierarchy is obtained by means of superposition of codes, previously given to all macroplaces to which the considered vertex belongs hierarchically. Taken as an example, the code of place *p2* is described as a product term *QP2 = Q1*/Q2*/Q3*.

6. PETRI NET MAPPING INTO LOGIC EXPRESSIONS

The logic controller is considered as an abstract reasoning system (rule-based system) implemented in reconfigurable hardware. The mapping between inputs, outputs, and local internal states of the system is described in a formal manner by means of logic rules (represented as sequents) with some temporal operators, especially with the operator "next" @[1,2]. As an example of a basic style of the textual controller description, the *transition-oriented declarative specification* is presented.

The declarative style of description is close to well-known production rules, which are principal forms of Petri net specification in LOGICIAN[1], CONPAR[6], PARIS[4,8], and PeNCAD[2,3]. It should be noted that here the names of transitions T1, T2, . . . , T8 serve only as decision rule labels, keeping the easy correspondence between Petri net transitions and their textual logic descriptions. The symbol |– denotes "yield," the symbol * stands for the logic conjunction operator, and / stands for negation.

```
T1[1,2] : P1[1,2] * X0  |-@P2[1] *@P4[2];
T2[1]   : P2[1] * X1  |-@P3[1];
T3[2]   : P4[2] * X3  |-@P5[2];
T4[1,2] : P3[1] * P5[1]   |-@P6 * @P7;
T5[1]   : P6[1] * X5*X6 |-@P8[1];
T6[2]   : P7[2] * /X2*/X4|-@P9[2];
T7[1]   : P8[1] * /X5 |-@P6[1];
T8[1,2] : P6[1] *P9[2] * /X6|-@P1[1,2].
```

The immediate combinational Moore-type output signals Y0–Y6, depend directly only on appropriate place markings:

```
P1[1,2]|-Y0; P2[1]|-Y1; P4[2]|-Y2; P7[2]|-Y3*Y4;
P8[1]|-Y5;P9[2] |-Y6.
```

Instead of combinational, intermediate Moore-type outputs Y0–Y6, the values of next registered outputs @Y0–@Y6 could be predicted in advance and included directly in the initial rule-based specification. The transition-oriented specification of the controller, after the substitution of encoding terms and next values of changing outputs, would look as follows:

```
T1: /Q1*X0 |- @Q1*@/Q2*@/Q3*@/Q4*/@Y0*@Y1*@Y2;
T2: Q1*/Q2*/Q3*X1 |- @Q1*@/Q2*@Q3*@/Y1;
(...)
T7: Q1*Q2*Q3*/X5 |-@Q1*@Q2*@/Q3*@/Y5;
T8: Q1*Q2*/Q3*Q4*/X6 |-@/Q1*@/Y6* @Y0.
```

In FPGA realizations of concurrent state machines with D flip-flops, it is worth introducing and directly implementing the intermediate binary signals *{T1, T2, . . . }* for detecting in advance the enabled transitions, which fire together with the next active edge of the clock. This way of design is especially suitable when relatively small FPGA macrocells might be easily reconfigured. In such Martin Bolton's style, the subset of simultaneously *activated transitions* keeps the logic signal 1 on its appropriate *transition status lines*. Simultaneously, the complementary subset of *blocked transitions* is recognized by logic signal 0 on its transition status lines. The great advantage of using transition status lines is the self-evident possibility of reducing the complexity of the next state and the output combinational logic. The registered output signals together with the next local state codes may be generated in very simple combinational structures, sharing together several common AND terms.

The simplified rule-based specification, especially planned for controllers with JK state and output registers, on the right side of sequents does not contain state coding signals, which keep their values stable, during the occurrence of transition[1]. Taking into account both the concept of transition status lines and introducing into specification only the changing registered Moore-type outputs signals, the specification may be rewritten as follows:

```
/Q1 * X0 |-T1;
            T1|-@Q1*@/Q2*@/Q3*@/Q4*/@Y0*@Y1*@Y2;
(...)
Q1*Q2*Q3*/X5 |-T7;
                T7 |- @/Q3*@/Y5;
Q1*Q2*/Q3*Q4*/X6 |-T8;
                    T8|- @/Q1*/@Y6*@Y0.
```

The translation of decision rules into VHDL is straightforward and can be performed as described in Refs. 2 and 6.

7. CONCLUSIONS

The paper presents the hierarchical Petri net approach to synthesis, in which the modular net is structurally mapped into field programmable logic. The hierarchy levels are preserved and related with some particular local state variable subsets. The proposed state encoding technique saves a number of macrocells and secures a direct mapping of Petri net into an FPL array. A concise, understandable specification can be easily locally modified.

The experimental Petri net to VHDL translator has been implemented on the top of standard VHDL design tools, such as ALDEC Active-HDL. VHDL syntax supports several conditional statements, which can be used to describe the topology and an interpretation of Petri nets.

ACKNOWLEDGMENT

The research was supported by the Polish State Committee for Scientific Research (Komitet Badań Naukowych) grant 4T11C 006 24.

REFERENCES

1. M. Adamski, Parallel controller implementation using standard PLD software. In: W.R. Moore, W. Luk (eds.), *FPGAs*. Abingdon EE&CS Books, Abingdon, England, pp. 296–304 (1991).
2. M. Adamski, SFC, Petri nets and application specific logic controllers. In: *Proc. of the IEEE Int. Conf. on Systems, Man, and Cybern.*, San Diego, USA, pp. 728–733 (1998).
3. M. Adamski, M. Wegrzyn (eds.), *Discrete-Event System Design DESDes'01*, Technical University of Zielona Gora Press Zielona Góra, ISBN: 83-85911-62-6 (2001).
4. K. Bilinski, M. Adamski, J.M. Saul, E.L. Dagless, Petri Net based algorithms for parallel controller synthesis. *IEE Proceedings-E, Computers and Digital Techniques*, **141**, 405–412 (1994).
5. R. David, H. Alla, *Petri Nets & Grafcet. Tools for Modelling Discrete Event Systems.* Prentice Hall, New York (1992).
6. J.M. Fernandes, M. Adamski, A.J. Proença, VHDL generation from hierarchical Petri net specifications of parallel controllers. *IEE Proceedings-E, Computer and Digital Techniques*, **144**, 127–137 (1997).
7. M. Heiner, Petri Net based system analysis without state explosion. In: *Proceedings of High Performance Computing'98*, April 1998, SCS Int., San Diego, pp. 394–403 (1988).
8. T. Kozlowski, E.L. Dagless, J.M. Saul, M. Adamski, J. Szajna, Parallel controller synthesis using Petri nets. *IEE Proceedings-E, Computers and Digital Techniques*, **142**, 263–271 (1995).
9. N. Marranghello, W. de Oliveira, F. Damianini, Modeling a processor with a Petri net extension for digital systems. In: *Proceedings of Conference on Design Analysis and Simulation of Distributed Systems–DASD 2004*, Part of the ASTC, Washington, DC, USA (2004).
10. T. Murata, Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, **77** (4), 541–580 (1989).
11. J.S. Sagoo, D.J. Holding, A comparison of temporal Petri net based techniques in the specification and design of hard real-time systems. *Microprocessing and Microprogramming*, **32**, 111–118 (1991).
12. M.E. Szabo (Ed.), *The collected papers of Gerhard Gentzen*. North-Holland Publishing Company, Amsterdam (1969).
13. A. Yakovlev, L. Gomes, L. Lavagno (eds.), *Hardware Design and Petri Nets*. Kluwer Academic Publishers, Boston (2000).
14. A.D. Zakrevskij, *Parallel Algorithms for Logical Control*. Institute of Engineering Cybernetics of NAS of Belarus, Minsk (1999) (Book in Russian).

Chapter 3

HIERARCHICAL PETRI NETS FOR DIGITAL CONTROLLER DESIGN

Grzegorz Andrzejewski
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgórna 50, 65-246 Zielona Góra, Poland; e-mail: G.Andrzejewski@iie.uz.zgora.pl

Abstract: This paper presents a model of formal specification of reactive systems. It is a kind of an interpreted Petri net, extended by important properties: hierarchy, history, and time dependencies. The syntax definition is introduced and the principles of graphical representation drawing are characterized. Semantics and dynamic behavior are shown by means of a little practical example: automatic washer controller.

Key words: reactive system; Petri net; formal specification.

1. INTRODUCTION

Reactive systems strongly interact with the environment. Their essence consists in appropriate control signals shaping in response to changes in communication signals. Control signals are usually called output signals, and communication signals input signals. It happens very frequently that a response depends not only on actual inputs but on the system's history too. The system is then called an automaton and its basic model is known as the *finite state machine* (FSM). But in a situation in which the system is more complicated, this model may be difficult to depict. The problem may be solved by using a hierarchy in which it is possible to consider the modeled system in a great number of abstraction layers. Such models as Statecharts or SyncCharts are the FSM expansions with a hierarchy[1,7,8].

Concurrency is a next important problem. Very often a situation occurs in which some processes must work simultaneously. In practice, it is realized by a net of related automata synchronized by internal signals. It is not easy to formally verify such a net because of necessity of separate analysis of each automaton[4,13].

A Petri net is a model in which concurrency is its natural property. Interpreted Petri nets are especially useful for modeling reactive systems. The existing apparatus of formal verification places this model very high among others in supporting concurrency. There exist many works in which the methodology of creating hierarchical nets is proposed[6,9,10,11,12,14]. But the models support only selected properties, such as the structure hierarchy or descriptions of time dependencies. In general they are too complicated for small microsystem realizations (object-oriented models).

In this paper, a different model of a hierarchical Petri net (HPN) is proposed, in which it is possible to describe strongly reactive systems at a digital microsystem realization platform. The model was partially described in Refs. 2 and 3.

2. SYNTAX

The following nomenclature is used: capital letters from the Latin alphabet represent names of sets, whereas small letters stand for elements of these sets. Small letters from the Greek alphabet represent functions belonging to the model. Auxiliary functions are denoted by two characteristic small letters from the Latin alphabet.

Def. 1 A hierarchical Petri net (HPN) is shown as a tuple:

$$\mathrm{HPN} = (P, T, F, S, \mathcal{T}, \chi, \psi, \lambda, \alpha, \varepsilon, \tau), \tag{1}$$

where

1. P is a finite nonempty set of places. In "flat" nets with places the capacity function $\kappa\colon P \to N \cup (\infty)$ describes the maximum number of tokens in place p. For reactive systems the function equals 1; for each place $p \in P$, $\kappa(p) = 1$.
2. T is a finite nonempty set of transitions. A sum of sets $P \cup T$ will be called a set of nodes and described by N.
3. F is a finite nonempty set of arcs, such that $F = F_o \cup F_e \cup F_i$, where $F_o\colon F_o \subset (P \times T) \cup (T \times P)$ and is called a set of ordinary arcs, $F_e\colon F_e \subset (P \times T)$ and is called a set of enabling arcs, and $F_i\colon F_i \subset (P \times T)$ and is called a set of inhibit arcs. In flat nets the weight function $\varpi : F \to N$ describes the maximum number of tokens that can be moved at the same time through arc f. For reactive systems $\forall f \in F, \varpi(p) \leq 1$. According to this, an extra-specification of arcs is possible: $\forall f \in F_o, \varpi(f) = 1; \forall f \in F_e \cup F_i, \varpi(f) = 0$.

4. S is a finite nonempty set of signals, such that $S = X \cup Y \cup L$, where X, Y, and L mean sets of input, output, and internal signals, respectively.
5. $\mathcal{T}$ is a discrete time scale, which is a set of numbers assigned to discrete values of time, sorted by an order relation.
6. $\chi\colon P \to 2^N$ is a hierarchy function, describing a set of immediate subnodes of place p. The expression χ^* denotes a transitive-reflexive closure of χ function, such that for each $p \in P$ the following predicates hold:

$$p \in \chi^*(p),$$
$$\chi(p) \in \chi^*(p),$$
$$p' \in \chi^*(p) \Rightarrow \chi(p') \subseteq \chi^*(p).$$

7. $\psi\colon P \to \{\mathit{true, false}\}$ is a Boolean history function, assigning a history attribute to every place p, such that $\chi(p) \neq \emptyset$. For basic places the function is not defined.
8. $\lambda\colon N \to 2^S$ is a labeling function, assigning expressions created from elements of set S to nodes from N. The following rules are suggested: places may be labeled only by subsets of $Y \cup L$ (a label *action* means an action assigned to a place); the label of transition may be composed of the following elements:
   cond – created on set $X \cup L \cup \{\mathit{false}, \mathit{true}\}$, being a Boolean expression imposed as a condition to transition t and generated by operators *not*, *or*, and *and*; the absence of *cond* label means *cond* = *true*;
   abort – created as a *cond* but standing in a different logical relation with respect to general condition for transition t enabling, represented graphically by # at the beginning of expression; absence of *abort* label means *abort* = *false*;
   action – created on set $Y \cup L$, meaning action assigned to transition t, represented graphically by / at the beginning of expression.
9. $\alpha\colon P \to \{\mathit{true, false}\}$ is an initial marking function, assigning the attribute of an initial place to every place $p \in P$. Initial places are graphically distinguished by a dot (mark) in circles representing these places.
10. $\varepsilon\colon P \to \{\mathit{true, false}\}$ is a final marking function, assigning the attribute of a final place to every place $p \in P$. Final places are graphically distinguished by $\times$ inside circles representing these places.
11. $\tau\colon N \to \mathcal{T}$ is a time function, assigning numbers from the discrete scale of time to each element from the set of nodes N.

The operation of a net is determined by movement of tokens. The rules of their movement are defined by conditions of transition enabling and action assigned to transition firing.

Let t_0 be an activation moment of node $n \in N$. The function $\tau(n)$ assigns a number to node n at the moment t_0: $\tau(n, t_0) = t$, where $t \in T$. In further instants the number is decremented, and after time t it accomplishes value 0: $\tau(n, t_0 + t) = 0$. The symbol $\tau(n) = 0$ describes the function at the moment, in which it equals 0.

Def. 2 P_p^{end} is a set of final places of a subnet assigned to macroplace p, such that

$$\forall_{p' \in \chi(p)} \varepsilon(p') = \text{true} \Rightarrow p' \in P_p^{\text{end}}. \tag{2}$$

Def. 3 $\xi: P \rightarrow P$ is a function of the set of final places, such that for macroplace p it returns its set of final places:

$$\xi(p) = P_p^{\text{end}}. \tag{3}$$

The expression ξ^* denotes a transitive-reflexive closure of ξ function, such that for each $p \in P$ and $\chi(p) \neq \emptyset$ the following predicates hold:

$$p \in \xi^*(p),$$
$$\xi(p) \in \xi^*(p),$$
$$p' \in \xi^*(p) \Rightarrow \xi(p') \subseteq \xi^*(p).$$

Def. 4 A final marking of a subnet is a marking that contains all final places of that subnet.

The conditions of transition *t* enabling:

$$\exists_{p \in P} \text{la}(t) = \text{p} \Rightarrow \text{ac}(p) = \text{true} \tag{4-a}$$

$$\forall_{p \in P_t^{\text{in(o)}} \cup P_t^{\text{in(e)}}} \text{ac}(p) = \text{true} \tag{4-b}$$

$$\forall_{p \in P_t^{\text{in(i)}}} \text{ac}(p) = \text{false} \tag{4-c}$$

$$\text{cond}(t) = \text{true} \tag{4-d}$$

$$\forall_{p \in P_t^{\text{in(o)}}} \chi(p) \neq \emptyset \Rightarrow \forall_{p' \in \xi^*(p)} (\text{ac}(p') = \text{true oraz } \tau(p') = 0) \tag{4-e}$$

$$\forall_{p \in P_t^{\text{in(o)}}} \tau(p) = 0 \tag{4-f}$$

$$\text{abort}(t) = \text{true} \tag{4-g}$$

Note: From all conditions the following logical expression can be composed (the general condition): $a^*b^*c^*(d^*e^*f + g)$, which means a possibility of enabling transition t without the need to satisfy conditions d, e, and f if g is true. This situation is known as *preemption*.

The actions assigned to transition *t* firing:

$$\forall_{p\in P_t^{\mathrm{in(o)}}} \mathrm{ac}(p) := \mathrm{false} \tag{5-a}$$

$$\forall_{p\in P_t^{\mathrm{in(o)}}} \chi(p) \neq \emptyset \Rightarrow \forall_{p'\in\chi^*(p)} \mathrm{ac}(p') := \mathrm{false} \tag{5-b}$$

$$\forall_{p\in P_t^{\mathrm{in(o)}}} \forall_{s\in\mathrm{action}(p)} s := \mathrm{false} \tag{5-c}$$

$$\forall_{p\in P_t^{\mathrm{in(o)}}} \chi(p) \neq \emptyset \Rightarrow \forall_{p'\in\overset{*}{\chi}(p)} \forall_{s\in\mathrm{action}(p')} s := \mathrm{false} \tag{5-d}$$

$$\tau(t) = 0 \Rightarrow \forall_{p\in P_t^{\mathrm{out}}} \mathrm{ac}(p) := \mathrm{true} \tag{5-e}$$

$$\forall_{p\in P_t^{\mathrm{out}}} \chi(p) \neq \emptyset \Rightarrow \left(\forall_{p'\in\chi^*(p)} \mathrm{ac}(\mathrm{la}(p')\right) = \mathrm{true\ oraz}$$
$$\psi(\mathrm{la}(p')) = \mathrm{false} \Rightarrow \mathrm{ac}(p') := \alpha(p')) \tag{5-f}$$

$$\forall_{p\in P_t^{\mathrm{out}}} \chi(p) \neq \emptyset \Rightarrow \left(\forall_{p'\in\chi^*(p)} \mathrm{ac}(\mathrm{la}(p')\right) = \mathrm{true} \wedge \psi(\mathrm{la}(p')) = \mathrm{true}$$
$$\wedge \exists_{p''\in\xi(\mathrm{la}(p'))} \mathrm{ac}(p'') = \mathrm{false} \Rightarrow \mathrm{ac}(p') := \mathrm{ac}(p', \not{t}_{\mathrm{e}})) \tag{5-g}$$

$$\forall_{p\in P_t^{\mathrm{out}}} \chi(p) \neq \emptyset \Rightarrow \left(\forall_{p'\in\chi^*(p)} \mathrm{ac}(\mathrm{la}(p')\right) = \mathrm{true} \wedge \psi(\mathrm{la}(p')) = \mathrm{true}$$
$$\wedge \forall_{p''\in\xi(\mathrm{la}(p'))} \mathrm{ac}(p'') = \mathrm{true} \Rightarrow \mathrm{ac}(p') := \alpha(p')) \tag{5-h}$$

$$\forall_{p\in P_t^{\mathrm{out}}} \forall_{s\in\mathrm{action}(p)} s := \mathrm{true} \tag{5-i}$$

$$\forall_{p\in P_t^{\mathrm{out}}} \chi(p) \neq \emptyset \Rightarrow \left(\forall_{p'\in\chi^*(p)} \mathrm{ac}(p') = \mathrm{true} \Rightarrow \forall_{s\in\mathrm{action}(p')} s := \mathrm{true}\right) \tag{5-j}$$

$$\tau(t, \iota_0) = \eta \Rightarrow \forall_{s\in\mathrm{action}(t)} s := \mathrm{true\ w\ przedziale} < \iota_0, \iota_0 + \eta + 1 > \tag{5-k}$$

where $\mathrm{ac}(p, \not{t}_{\mathrm{e}})$ is the state of possession (or not) of a token by place p at an instant, in which the token left place la(p).

Note: Actions *e–j* are performed when $\tau(t) = 0$. Action k is performed during all activity time of transition t.

The most important ideas are defined additionally:

Let be given a hierarchical Petri net and place p from the set of places of this net.

Def. 5 A place p is called a basic place if $\chi(p) = \emptyset$.

Def. 6 A place p is called a macroplace if it isn't a basic place: $\chi(p) \neq \emptyset$.

Def. 7 A set of input places of transition t is called $P_t^{\mathrm{in(o)}}$, such that $P_t^{\mathrm{in(o)}} = \{p \in P : (p, t) \in F_\mathrm{o}\}$.

Def. 8 A set of enabling places of transition t is called $P_t^{\mathrm{in(e)}}$, such that $P_t^{\mathrm{in(e)}} = \{p \in P : (p, t) \in F_\mathrm{e}\}$.

Def. 9 A set of prohibit places of transition t is called $P_t^{\mathrm{in(i)}}$, such that $P_t^{\mathrm{in(i)}} = \{p \in P : (p, t) \in F_\mathrm{i}\}$.

Def. 10 A set of output places of transition t is called P_t^{out}, such that $P_t^{\mathrm{out}} = \{p \in P : (t, p) \in F_\mathrm{o}\}$.

Def. 11 The function ac: $P \rightarrow \{\mathit{true}, \mathit{false}\}$ is called a place activity function and it assigns *true* to each place that has a token, and *false* otherwise.

Def. 12 The place p is the lowest ancestor of node n', such that $n' \in \chi(p)$, which is described by $\mathrm{la}(n') = p$.

Def. 13 Let N^i be a set of nodes assigned to a macroplace p by the hierarchy function $N^\mathrm{i} = \chi(p)$, and let F^i be a set of all arcs joining the nodes belonging to N^i. Then Z^i is a subnet assigned to macroplace p, such that $Z^\mathrm{i} = N^\mathrm{i} + F^\mathrm{i}$. All subnets assigned to macroplaces are required to be disjoint (no common nodes and arcs).

3. SEMANTICS

3.1 Synchronism

One of the basic assumptions accepted in the HPN model is synchronism. Changes of internal state of a net follow as a result of inputs changes in strictly appointed instants given by discrete scale of time. It entails a possibility of a simultaneous execution of more than one transition (if the net fulfills the persistent property). Additionally, it makes possible to simplify formal verification methods and practical realization of the model[5].

3.2 Hierarchy

The hierarchy property is realized by a net decomposition, in which other nets are coupled with distinguished places. These places are called macroplaces, and the assigned nets are called subnets. A subnet is a basic net if no macroplaces are assigned to it. A subnet is a final net if it contains no macroplaces. Marking of a macroplace is an activate condition of the corresponding subnet. Such a concept allows not only much clearer projects with high complication structures but also testing of selected behavioral properties for each subnet separately.

3.3 History

Often a situation occurs in which an internal state on selected hierarchy levels must be remembered. In an HPN it is realized throughout, ascribing the history attribute $\{H\}$ to a selected macroplace. With a token leaving macro, all token locations in the adequate subnet are remembered. And after renewed macro activation, tokens are inserted to lately active places.

For the convenience of the user, a possibility of ascribing the history attribute to all subordinated nets is included. Operator $\{H^*\}$ is used.

3.4 System reaction

There exists a possibility of assigning labels to the set of nodes. The system is able to perform given actions described by the labels; e.g., testing input conditions on the input signals set and setting (or clearing) output signals in appropriate time moments.

3.5 Time parameters

Time parameters are associated with nodes (see function $\tau(n)$ in Def. 1). Ascribing time $\not{t}$ from the discrete scale of time to place p determines the minimum activity time of this place. This means that the output transition of place p will be enabled only after time $\not{t}$, beginning at the moment of activation of place p. The time parameter associated with transition determines the activity time of this transition, and it means that after removing tokens from all input places of transition t, insertion of tokens to all output places ensues only after time $\not{t}$. This solution provides in practice great possibilities in describing strongly time dependent systems.

3.6 Graphical representation

An important feature of the model is its user-friendly graphical representation. In general, it is an oriented graph with two kinds of nodes connected by arcs. On the basic layer, graphical symbols used in an HPN are the same as in the flat interpreted Petri nets. Places are represented by arcs, transitions by thick beams, and markers (tokens) by dots inside the places. The macroplace and its expansion are presented in Fig. 3-1. Macroplace MP has a deep history attribute. The subnet assigned to MP is a compound of macroplace P1 and basic

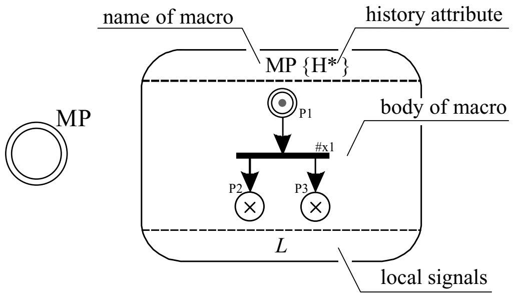


Figure 3-1. A macroplace with assigned expansion.

places P2 and P3. P1 is an initial place, and places P2 and P3 are final places. Macroplace P1 can be deprived of activity by means of *abort-condition* $\times 1$.

There is a simple example showing a simplified control system of initial washing in an automatic washer (Fig. 3-2).

After turning on the washing program, valve V1 is opened and water is infused. The infusing process lasts to the moment of achieving L1 level. At the same time, after exceeding L2 level (total sinking of heater H) if the temperature is below the required (TL1), the heater system is turned on. After closing valve V1 the washing process is started, in which the washing cylinder is turned alternately left and right for 10 sec. with a 5-sec. break. The process of keeping the temperature constant is active for the whole washing cycle. The cycle is turned off after 280 sec. and the cylinder is stopped, the heater is turned off, and valve V2 is opened for water removal.

There is a possibility of describing such a system by means of hierarchical Petri nets (Fig. 3-3).

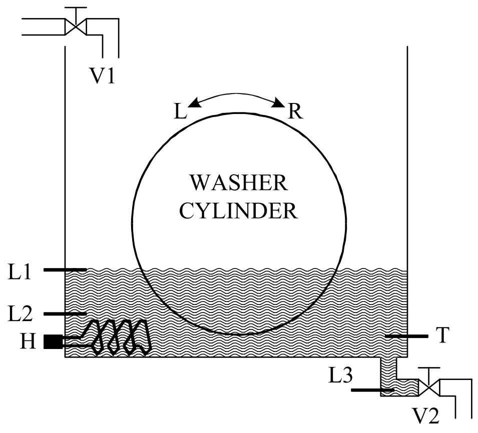


Figure 3-2. An example of an automatic washer control system.

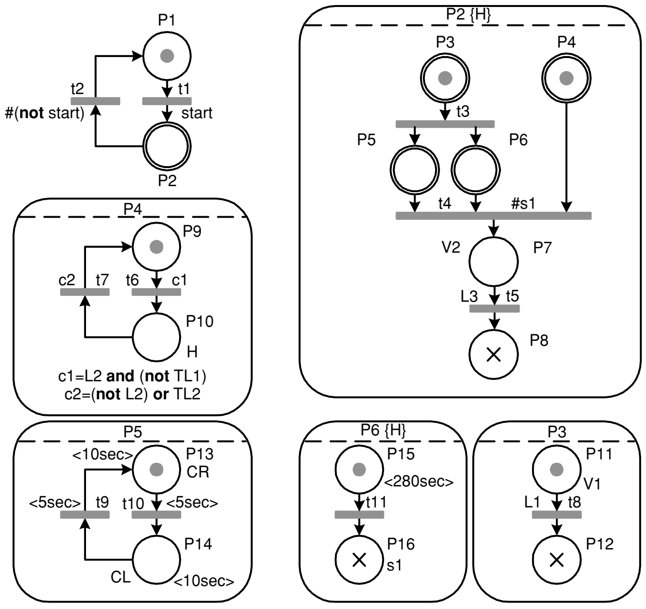


Figure 3-3. Fragment of the net modeling the washer controller.

4. CONCLUSION

The model offers a convenient means for a formal specification of reactive systems. An equivalent model of textual notation (HPN format) is worked out too. The rules of assigning the external function (e.g., ANSI C or VHDL) to nodes of the net are the subject of research.

Further work and research shall focus on creating tools for automatic analysis and synthesis of the model on a hardware/software codesign platform in an ORION software package developed by the author.

ACKNOWLEDGMENT

The research was supported by the Polish State Committee for Scientific Research (Komitet Badań Naukowych) grant 4T11C 006 24.

REFERENCES

1. C. André, Synccharts: A visual representation of reactive behaviors. Technical Report RR 95-52, I3S, Sophia-Antipolis, France (1995).

2. G. Andrzejewski, Hierarchical Petri Net as a representation of reactive behaviors. In: *Proceedings of International Conference Advanced Computer Systems: ACS'2001*, Szczecin, Polska, Part 2, pp. 145–154 (2001).
3. G. Andrzejewski, *Program Model of Interpeted Petri Net for Digital Microsystems Design*. University of Zielona Góra Press, Poland (2003) (in Polish).
4. F. Balarin et. al., *Hardware–Software Co-Design of Embedded Systems. The POLIS Approach*. Kluwer Academic Publishers (1999).
5. Z. Banaszak, J. Kuś, M. Adamski, *Petri Nets, Modeling, Control and Synthesis of Discrete Systems*. WSI Press (1993) (in Polish).
6. M.P.J. Bolton, *Digital Systems Design with Programmable Logic*. Addison-Wesley Publishing Company, Wakingham (1990).
7. D.D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*. Prentice-Hall, Englewood Cliffs, NJ (1994).
8. D. Harel, Statecharts: A Visual Formalism for Complex Systems, North-Holland, Amsterdam. Science of Computer Programming, **8** 231–274 (1987).
9. T. Holvoet, P. Verbaeten, Petri charts: An alternative technique for hierarchical net construction. In: *IEEE Conference on Systems, Man and Cybernetics* (1995).
10. J.E. Hong, D.H. Bae, HOONets: Hierarchical object-oriented Petri nets for system odeling and analysis. KAIST Technical Report CS/TR-98-132 (1998).
11. K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science, Springer-Verlag (1997).
12. C. Johnsson, K.E.Årzen, High-level grafcet and batch control. In: *Conference Proceedings ADPM'94 Automation of Mixed Processes: Dynamical Hybrid Systems*, Bryssel (1994).
13. G. de Micheli, *Synthesis and optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.
14. S. Schof, M. Sonnenschein, R. Wieting, High-level modeling with THORNs. In: *Proceedings of 14th International Congress on Cybernetics*, Namur, Belgium (1995).

Section II

Analysis and Verification of Discrete-Event Systems

Chapter 4

WCET PREDICTION FOR EMBEDDED PROCESSORS USING AN ADL

Adriano Tavares, Carlos Silva, Carlos Lima, José Metrolho, and Carlos Couto
Department of Industrial Electronics, University of Minho, Campus de Azurém, 4800-058 Guimarães, Portugal; e-mail: atavares@del.uminho.pt, csilva@del.uminho.pt, clima@del.uminho.pt, jmetrolho@del.uminho.pt, ccouto@del.uminho.pt

Abstract: A method for analyzing and predicting the timing properties of a program fragment will be described. First an architectural description language implemented to describe a processor's architecture is presented, followed by the presentation of a new, static worst-case execution time (WCET) estimation method. The timing analysis starts by compiling a processor's architecture program, followed by the disassembling of the program fragment. After sectioning the assembler program into basic blocks, call graphs are generated and these data are later used to evaluate the pipeline hazards and cache miss that penalize the real-time performance. Some experimental results of using the developed tool to predict the WCET of code segments using some Intel microcontroller are presented. Finally, some conclusions and future work are presented.

Key words: architectural description language (ADL); worst-case execution time (WCET); language paradigm; timing scheme, timing analysis.

1. INTRODUCTION

Real-time systems are characterized by the need to satisfy a huge timing and logical constraints that regulate their correctness. Therefore, predicting a tight worst-case execution time (WCET) of a code segment will be a must to guarantee the system correctness and performance. The simplest approach to estimate the execution time of a program fragment for each arithmetic instruction is to count the number of times it appears on the code, express the contribution of this instruction in terms of clock cycles, and update the total clock cycles with this contribution. Nevertheless, these approaches are unrealistic since they ignore the system interferences and the effects of cache and pipeline, two very important features of some processors that can be used in our hardware architecture. Some very elaborated methodologies for WCET estimation, such as Shaw[1], were developed in the past, but none of them takes into account the effects of cache and pipeline.

Theoretically, the estimation of WCET must skip over all the profits provided by modern processors, such as caches and pipeline (i.e., each instruction execution suffers from all kind of pipeline hazards and each memory access would cause a cache miss), as they are the main source of uncertainty. Experimentally, a very pessimistic result would be obtained, thus making useless these processors' resources. Some WCET estimation schemes oriented to modern hardware features were presented in the past few years, and among them we refer to Nilsen[2], Steven Li[3], Whalley[4], and Sung-Soo Lim[5]. However, these WCET estimators do not address some specificity of our target processors (microcontrollers and Digital Signal Processors (DSPs)), since they are oriented to general-purpose processors. Therefore, we propose a new machine-independent predictor, implemented as an ADL for processor description. Such a machine-independent scheme using an ADL was used before by Tremblay[6] to generate machine-independent code, by Proebsting and Fraser[7] to describe pipeline architectures, and by Nilsen[5] to implement a compiler, a simulator, and a WCET estimator for pipeline processors.

2. ADL FOR EMBEDDED PROCESSOR

The purpose of any little language, typically, is to solve a specific problem and, in so doing, simplify the activities related to the solution of the problem. Our little language is an ADL, so its statements are created on the basis of the tasks that must be performed to describe a processor's architectures in terms of structure and functional architecture of the interrupt controller, PTS (peripheral transaction server), PWM (pulse width modulation), WG (waveform generator), and HIS (high-speed input), instruction set, instruction semantics, addressing modes, processor's registers, instruction coding, compiler's specificity, scratch-pad memory, pipeline and cache resources, and many others specific features of an embedded processor. For this ADL, we adopt a procedural and modular paradigm (language paradigm defines how the language processor must process the built-in statements), such that modules are independent of each other. The sequence of modules execution does not matter, but the register module must always be the first to be executed, and within each module an exact sequence of instructions is specified and the computer executes them in the specified order. An ADL program describing a processor is written by modules,

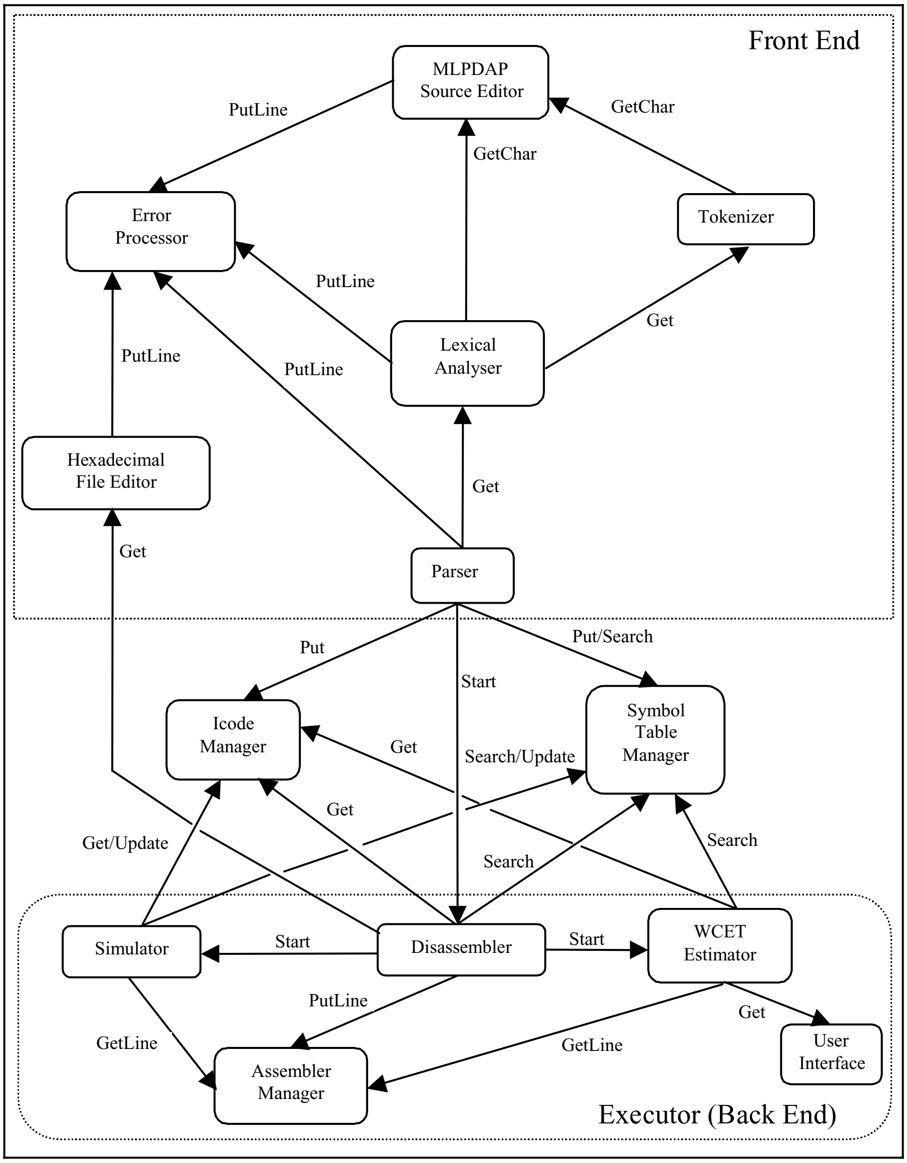


Figure 4-1. Organization of the ADL processor language.

each describing a specific processor's feature such as instruction set, interrupt structure and mechanism, registers structure, memory organization, pipeline, data cache, instruction cache, PTS, and so on. A module can be defined more than once, and it is a processor language (Fig. 4-1) job to verify the information consistency among them and concatenate all them into a single module.

The disassembling process consists of four phases and has as input an executable file containing the code segment that one wants to measure and the compiled version of an ADL program. The disassembling process starts at the start-up code address (start-up code is the bootstrap code executed immediately after the reset or power-on of the processor) and follows the execution flow of the program:

1. Starting at the start-up code address, it follows all possible execution paths till reaching the end address of the "main" function. At this stage, all function calls are examined and their entry code addresses are pushed into an auxiliary stack.
2. From the entry address of the "main" function, it checks the main function code for interrupt activation.
3. For each active interrupt, it gets its entry code address and pushes it into the auxiliary stack.
4. It pops each entry address from the auxiliary stack and disassembles it, following the function's execution paths.

The execution of the simulation module is optional and the associated process is described by a set of operation introduced using a function named "SetAction." For instance, the simulation process including the flag register affectation, associated to an instruction, is described using SetAction calls to specify a sequence of operations. Running the simulation process before the estimation process will produce a more optimistic worst-case timing analysis since it can

1. rectify the execution time of instructions that depend on data locations, such as stack, internal, or external memory;
2. solve the indirect address problem by checking if it is a jump or a function call (function call by address);
3. estimate the iteration number of a loop.

The WCET estimator module requires a direct interaction with the user as some parameters are not directly measurable through the program code. Note that the number of an interrupt occurrence and the preview of a possible maximum iterations number associated with an infinite loop are quite impossible to be evaluated using only the program code. The WCET estimation process is divided into two phases:

1. First, the code segment to be measured is decomposed into basic blocks;
2. For each basic block, the lower and upper execution times are estimated using the shortest path method and a timing scheme[1].

The shortest path algorithm with the basic block graph as input is used to estimate the lower and the upper bounds on the execution time of the code segment. For the estimation of the upper bound, the multiplicative inverse of the upper execution time of each basic block is used. A basic block is a sequence of assembler's instructions, such as "only the first instruction can be prefixed by a label" and "only the last one can be a control transfer instruction."

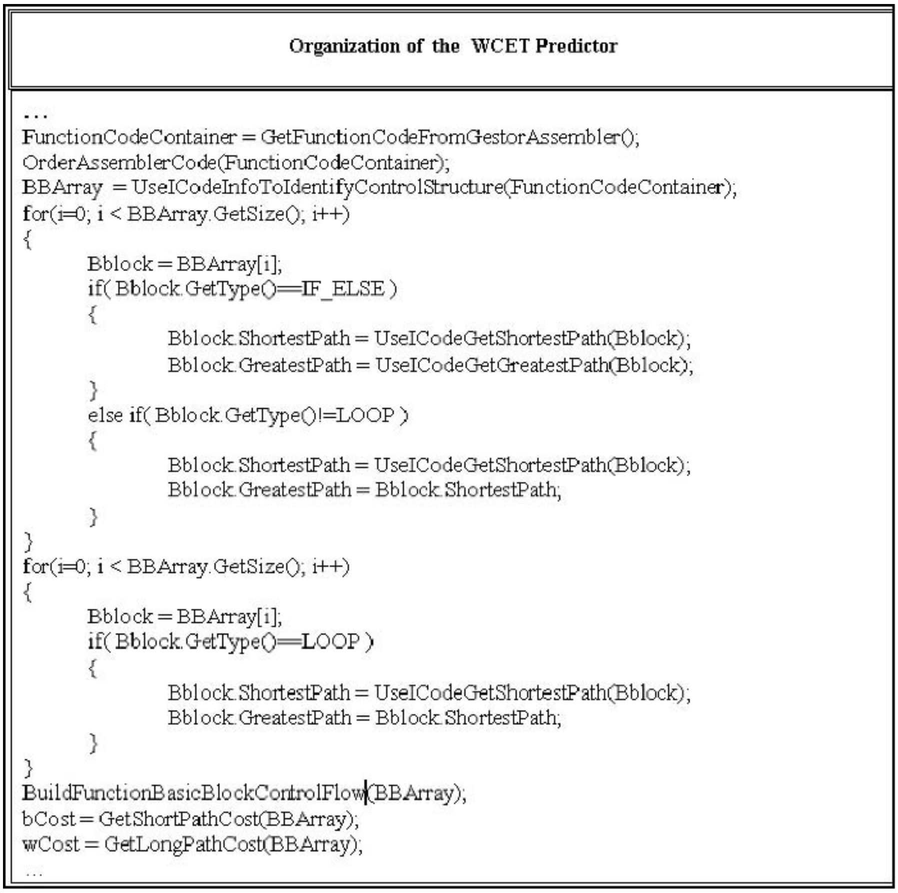
The decomposition phase is carried out following the steps given below:

1. rearrangement of code segment to guarantee the visual cohesion of a basic block (note that the ordering of instructions by address makes the visualization of the inter basic block control flow more difficult, because of long jump instructions that can occur between basic blocks. To guarantee visual cohesion, all sequence of instructions are rearranged by memory address, excluding those located from long jump labels, which are inserted from the last buffer index);
2. characterization of the conditional structure through the identification of the instructions sequence that compose the "if" and "else" body;
3. characterization of the loop structure through the identification of the instructions sequence that composes the loop body, control, and transfer control (it is essential to discern between "while/for" and "do while" loops since the timing schemes are different);
4. building a basic block graph showing all the execution paths between basic blocks;
5. finding the lower and upper execution time for each basic block.

2.1 Pipeline Modeling

The WCET estimator presented so far (Fig. 4-2) considers that an instruction's execution is fixed over the program execution; i.e., it ignores the contribution of modern processors. Note that the dependence among instruction can cause pipeline hazards, introducing a delay in the instruction execution. This dependence emerges as several instructions are simultaneously executed, and as a result of this parallel execution among instructions, the execution time of an instruction fluctuates depending on the set of its neighboring instructions. Our ADL's processor language analyzes the pipeline using the pipeline hazard detection technique (Fig. 4-3) suggested by Proebsting and Fraser[7]. The ADL models the pipeline as a set of resources and each instruction as a process that acquires and consumes a subset of resources for its execution. Special-purpose functions, such as "setPipeStage(Mn)" and "SetPipeFunctionalUnit(Mn, num)," are used to define the pipeline stages and functional units, respectively. For each instruction, there is a set of functions to solve the following points:

1. *Instr.SetSourceStage(s_Opr, Stg)* specifies the pipeline stage, at which each source operand must be available,



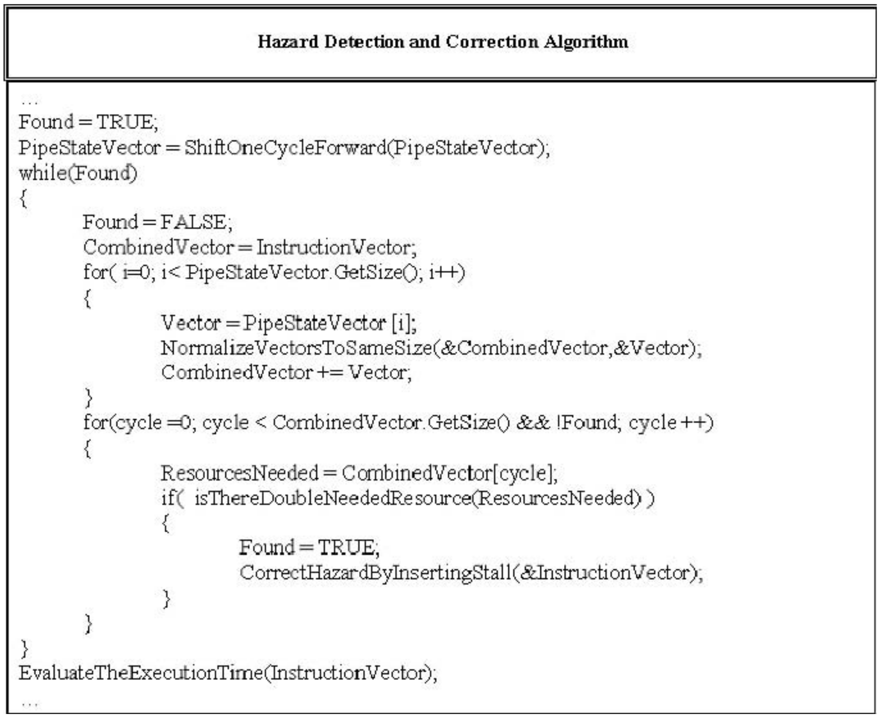
Organization of the WCET Predictor

```
...
FunctionCodeContainer = GetFunctionCodeFromGestorAssembler();
OrderAssemblerCode(FunctionCodeContainer);
BBArray = UseICodeInfoToIdentifyControlStructure(FunctionCodeContainer);
for(i=0; i < BBArray.GetSize(); i++)
{
        Bblock = BBArray[i];
        if( Bblock.GetType()==IF_ELSE )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = UseICodeGetGreatestPath(Bblock);
        }
        else if( Bblock.GetType()!=LOOP )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = Bblock.ShortestPath;
        }
}
for(i=0; i < BBArray.GetSize(); i++)
{
        Bblock = BBArray[i];
        if( Bblock.GetType()==LOOP )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = Bblock.ShortestPath;
        }
}
BuildFunctionBasicBlockControlFlow(BBArray);
bCost = GetShortPathCost(BBArray);
wCost = GetLongPathCost(BBArray);
...
```

Figure 4-2. Organization of the WCET predictor.

2. *Instr.SetResultStage(d_Opr, Stg)* specifies the pipeline stage, at which the output of the destination operand becomes available with the output of the destination operand,
3. *Instr.SetStageWCET(stg, tm)* specifies each pipeline stage required to execute an instruction and the execution time associated with that stage,
4. *Instr.SetbranchDelayCost(tm)* sets the control hazard cost associated with a branch instruction.

The pipeline analysis of a given basic block must always take into account the influences of the predecessor basic blocks (note that the dependence among instructions can cause pipeline hazards, introducing a delay in the instructions execution); otherwise, it leads to an underestimation of the execution time. Therefore, at the hazard detection stage of a given basic block, it will always



Hazard Detection and Correction Algorithm

```
...
Found = TRUE;
PipeStateVector = ShiftOneCycleForward(PipeStateVector);
while(Found)
{
        Found = FALSE;
        CombinedVector = InstructionVector;
        for( i=0; i< PipeStateVector.GetSize(); i++)
        {
                Vector = PipeStateVector [i];
                NormalizeVectorsToSameSize(&CombinedVector,&Vector);
                CombinedVector += Vector;
        }
        for(cycle =0; cycle < CombinedVector.GetSize() && !Found; cycle ++)
        {
                ResourcesNeeded = CombinedVector[cycle];
                if( isThereDoubleNeededResource(ResourcesNeeded) )
                {
                        Found = TRUE;
                        CorrectHazardByInsertingStall(&InstructionVector);
                }
        }
}
EvaluateTheExecutionTime(InstructionVector);
...
```

Figure 4-3. Hazard detection and correction algorithm based on Proebsting's technique.

incorporate the pipeline's state associated with the predecessor basic blocks over the execution paths. The resources vector that describes the pipeline's state will be iteratively updated by inserting pipeline stalls to correct the data and/or structural hazards when the next instruction is issued. If these two hazards happen simultaneously, the correction process starts at the hazard that occurred first and then it will be checked whether the second still remains. The issuing of a new instruction will be always preceded by the updating of the previous pipeline's state, achieved by shifting the actual pipeline resource vector one cycle forward (Fig. 4-4).

The pipeline architectures, usually, present special techniques to correct the execution flow when a control hazard happens. For instance, the delay transfer control technique offers the hardware an extra machine cycle to decide the branch. Also, special hardware is used to determine the branch label and value condition at the end of the instruction's decode. As one can conclude, the execution of delay instructions does not depend on the branch decision, and it is always carried out. Therefore, we model the control hazard as being caused by all kinds of branch instruction and by adding the sum of

Organization of WCET Predictor with Pipeline Effect

```
...
EmptyPipelineResourceVector();
BBlock= GetBBlockWithNoPredecessor();
while(BBlock = = VALID)
{
        SuccBBlock = BBlock;
        FatherPipeState = ShiftOneCycleForward(PipeStateVector);
        while(SuccBBlock = = VALID)
        {
                PipeStateVector = FatherPipeState;
                for(i=0; i< SuccBBlock.GetSize(); i++)
                {
                        InstructionICode = IdentifyICodeOfInstruction(SuccBBlock [i]);
                        DataHVector = InstructionICode.GetResourceNeeded();
                        ControlHVector = InstructionICode.GetResourceNeeded();
                        DetectandCorrectHazard(ControlHVector, DataHVector);

                }
                SuccBBlock = NextSuccessorBBlock(BBlock);
        }
        BBlock= NextBBlockReadyToEvaluation();
        PipeStateVector = GetPipelineStateOfFather(BBlock);
}
CallNormalWCETEstimator();
...
```

Figure 4-4. Organization of the WCET predictor with pipeline effect.

execution time of all instructions in the slot delay to the basic block execution time.

2.2 Cache and scratchpad memory modeling

Cache is a high-speed and small-sized memory, typically a SRAM that contains parts of the most recent accesses to the main memory. Nowadays, the time required to load an instruction or data to the processor is much longer than the instruction execution time. The main function of a cache memory is to reduce the time needed to move the information from and to the processor. An explanation for this improvement comes from the locality of reference theory – at any time, the processor will access a very small and localized region of the main memory, and the cache loads this region, allowing faster memory accesses to the processor.

In spite of the memory performance enhancement, the cache makes the execution time estimation harder, as the execution time of any instruction will vary and depend on the presence of the instruction and data into the caches. Furthermore, to exactly know if the execution of a given instruction causes a cache miss/hit, it will be necessary to carry out a global analysis of the program. Note that an instruction's behavior can be affected by memory references that happened a long time ago. Adversely, the estimation of WCET becomes harder for the modern processors, as the behaviors of cache and pipeline depend on each other. Therefore, we propose the following changes to the algorithm that takes into account the pipeline effects:

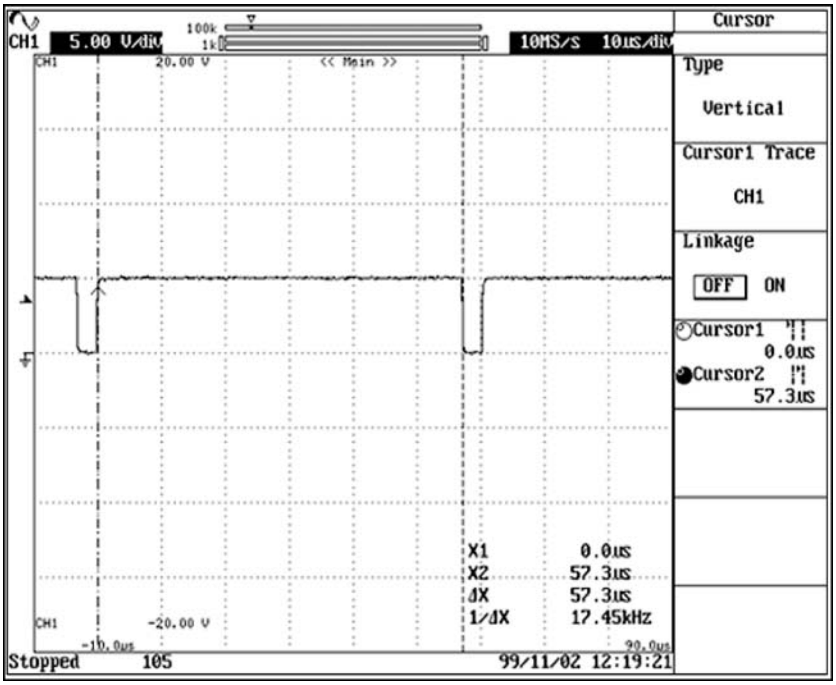
1. Classify the cache behavior[4] for any data and instruction as cache hit or cache miss before the analysis of the pipeline behavior.
2. Before the issuing of an instruction, verify if there is any cache miss related to the instruction; if there is, first apply the miss penalty and then the detection and correction of pipeline hazards.

In contrast, the scratchpad memory that is a software-controlled on-chip memory located in a separate address region provides predictable execution time. In a scratchpad memory system, the mapping of programs elements is done statically during compile time either by the user or automatically by the compiler. Therefore, whereas an access to cache SRAM is subject to compulsory, capacity and conflict misses, the scratchpad memory guarantees a single-cycle access time, and it can be modeled like any internal memory access, using the access time as the only parameter.

3. EXPERIMENTAL RESULTS

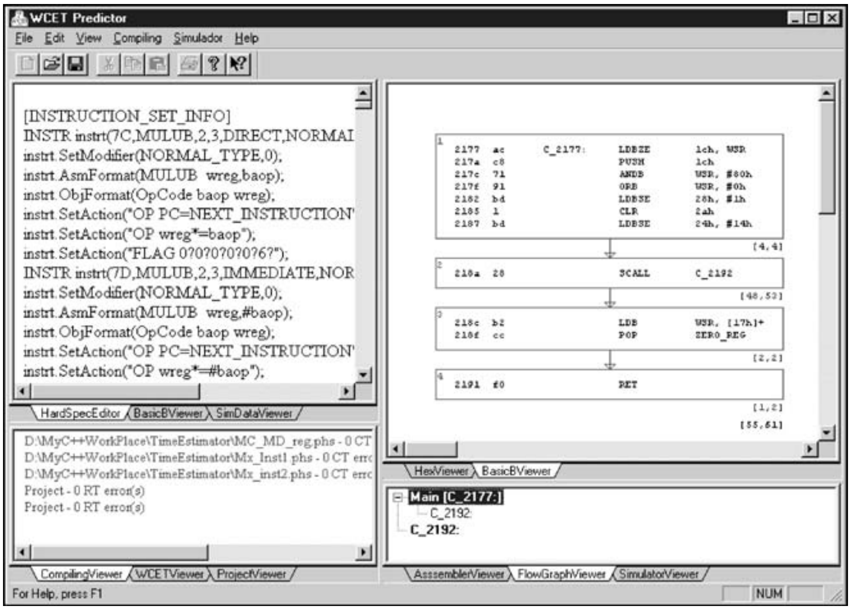
For the moment, we will present some results using the 8 × C196 Intel microcontrollers as they are the only ones present with all the needed execution time information in the user's guide. But we hope to present soon the results of experiments with modern processors such as Texas Instruments DSPs, Intel 8 × C296, PICs, and so on. Figure 4-5 shows the result achieved by a direct measurement of a program composed by two functions: main() and func(). This program was instrumented to allow a direct measurement with a digital oscilloscope through pin 6 of port 2 (P2.6).

At a first stage, the WCET estimator builds the call graph given at the lower right quadrant of Fig. 4-6 and then func(), identified by the label C_2192, is processed, providing a similar screen (Fig. 4-7). At the upper right quadrant, information, such as execution time of individual basic blocks, basic block control flow, and function execution time, is presented. The lower right quadrant can present the assembly code translated by the disassembler from the executable code, the call graph, and the simulator state. The upper left quadrant



```
X1      0.0us
X2      57.3us
dX      57.3us
1/dX    17.45kHz
```

Figure 4-5. Direct measurement of an instrumented program using a digital oscilloscope to monitor pin 2 of port P2 (P2.6).


WCET Predictor
File Edit View Compiling Simulador Help
HardSpecEditor BasicBViewer SimDataViewer
HexViewer BasicBViewer
CompilingViewer WCETViewer ProjectViewer
AsssemblerViewer FlowGraphViewer SimulatorViewer
For Help, press F1
NUM

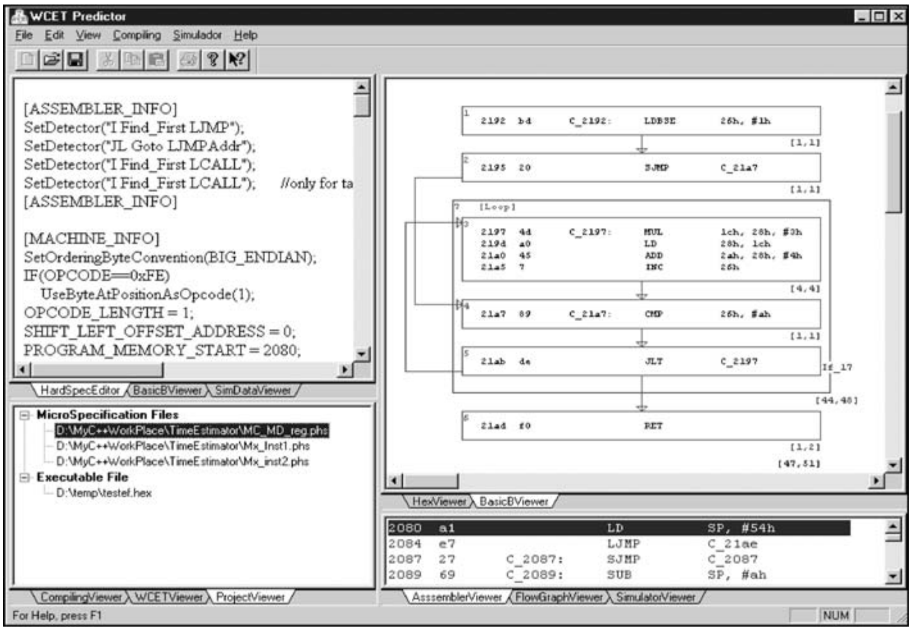
```
[INSTRUCTION_SET_INFO]
INSTR instrt(7C,MULUB,2,3,DIRECT,NORMAI
instrt.SetModifier(NORMAL_TYPE,0);
instrt.AsmFormat(MULUB  wreg,baop);
instrt.ObjFormat(OpCode baop wreg);
instrt.SetAction("OP PC=NEXT_INSTRUCTION'
instrt.SetAction("OP wreg*=baop");
instrt.SetAction("FLAG 0?0?0?0?0?6?");
INSTR instrt(7D,MULUB,2,3,IMMEDIATE,NOR
instrt.SetModifier(NORMAL_TYPE,0);
instrt.AsmFormat(MULUB  wreg,#baop);
instrt.ObjFormat(OpCode baop wreg);
instrt.SetAction("OP PC=NEXT_INSTRUCTION'
instrt.SetAction("OP wreg*=#baop");
```

```
D:\MyC++WorkPlace\TimeEstimator\MC_MD_reg.phs - 0 CT
D:\MyC++WorkPlace\TimeEstimator\Mx_Inst1.phs - 0 CT errc
D:\MyC++WorkPlace\TimeEstimator\Mx_inst2.phs - 0 CT errc
Project - 0 RT error(s)
Project - 0 RT error(s)
```

```
1
2177  ac     C_2177:    LDBZE    1ch, WSR
217a  c8                PUSH     1ch
217c  71                ANDB     WSR, #80h
217f  91                ORB      WSR, #0h
2182  bd                LDBSE    28h, #1h
2185  1                 CLR      2ah
2187  bd                LDBSE    24h, #14h
                                          [4,4]
2
218a  28                SCALL    C_2192
                                          [48,53]
3
218c  b2                LDB      WSR, [17h]+
218f  cc                POP      ZERO_REG
                                          [2,2]
4
2191  f0                RET
                                          [1,2]
                                          [55,61]
```

```
Main [C_2177:]
  C_2192:
C_2192:
```

Figure 4-6. WCET = 61 μs was estimated for the code segment measured in Fig. 4-4.


WCET Predictor
File Edit View Compiling Simulador Help
HardSpecEditor BasicBViewer SimDataViewer
CompilingViewer WCETViewer ProjectViewer
HexViewer BasicBViewer
AsssemblerViewer FlowGraphViewer SimulatorViewer
For Help, press F1
NUM

```
[ASSEMBLER_INFO]
SetDetector("I Find_First LJMP");
SetDetector("JL Goto LJMPAddr");
SetDetector("I Find_First LCALL");
SetDetector("I Find_First LCALL");     //only for ta
[ASSEMBLER_INFO]

[MACHINE_INFO]
SetOrderingByteConvention(BIG_ENDIAN);
IF(OPCODE==0xFE)
   UseByteAtPositionAsOpcode(1);
OPCODE_LENGTH = 1;
SHIFT_LEFT_OFFSET_ADDRESS = 0;
PROGRAM_MEMORY_START = 2080;
```

```
MicroSpecification Files
   D:\MyC++WorkPlace\TimeEstimator\MC_MD_reg.phs
   D:\MyC++WorkPlace\TimeEstimator\Mx_Inst1.phs
   D:\MyC++WorkPlace\TimeEstimator\Mx_inst2.phs
Executable File
   D:\temp\testef.hex
```

```
1
2192  bd     C_2192:    LDBSE     26h, #1h
                                              [1,1]
2
2195  20                SJMP      C_21a7
                                              [1,1]
7  [Loop]
3
2197  4d     C_2197:    MUL       1ch, 28h, #3h
219d  a0                LD        28h, 1ch
21a0  45                ADD       2ah, 28h, #4h
21a5  7                 INC       26h
                                              [4,4]
4
21a7  89     C_21a7:    CMP       26h, #ah
                                              [1,1]
5
21ab  de                JLT       C_2197      If_17
                                              [44,48]
6
21ad  f0                RET
                                              [1,2]
                                              [47,51]
```

```
2080   a1                    LD        SP, #54h
2084   e7                    LJMP      C_21ae
2087   27         C_2087:    SJMP      C_2087
2089   69         C_2089:    SUB       SP, #ah
```

Figure 4-7. WCET analysis of the function denominated func().

presents parts of the ADL program describing the microcontroller architecture.

4. CONCLUSIONS

A very friendly tool for WCET estimation was developed, and the results obtained over some Intel microcontroller were very satisfactory. To complete the evaluation of our tool, we will realize more tests using other classes of processors, such as DSPs, PICs, and some Motorola microcontrollers. A plenty use of this tool requires some processor information, such as the execution time of each instruction composing the processor instruction set, sometimes not provided in the processor user's guide. In such a case, to time an individual instruction, we recommended the use of the logic analyzer to trigger on the opcode at the target instruction location and on the opcode and location of the next instruction.

On the basis of previous studies of some emergent ADLs[8,9], we are developing a framework to generate accurate simulators with different levels of abstraction by using information embedded in our ADL. With this ADL it will be possible to generate simulators, and other tools, for microprocessors with different features such as superpipelining, superscalar with out-of-order execution, VLIW, and so on.

REFERENCES

1. Alan C. Shaw, Deterministic timing schema for parallel programs, Technical Report 90-05-06, Department of Computer Science and Engineering, University of Washington, Seattle (1990).
2. K. Nilsen, B. Rygg, Worst-case execution time analysis on modern processor. *ACM SIGPLAN Notices*, **30** (11), 20–30 (1995).
3. Y. Steven Li et al., Efficient microarchitecture modeling and path analysis for real-time software. Technical Report, Department of Electrical Engineering, Princeton University.
4. C. Healy, M. Harmon, D. Whalley, Integrating the timing analysis of pipelining and instruction caching. Technical Report, Computer Science Department, Florida State University.
5. Sung-Soo Lim, C. Yun Park et al., An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, **21** (7), 593–604 (1995).
6. P. Sorenson, J. Tremblay, *The Theory and Practice of Compiler Writing*. McGraw-Hill, New York, ISBN 0-07-065161-2 (1987).
7. C.W. Fraser, T. Proebsting, Detecting pipeline structural hazards quickly. In: *Proceedings of the 21th Annual ACM SIGPLAN_SIGACT Symposium on Principles of Programming Languages*, pp. 280–286 (January 1994).
8. S. Pees et al. LISA – Machine description language for cycle-accurate models of programmable DSP architectures. In: *ACM/IEEE Design Automation Conference* (1999).
9. Ashok Halambi, et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: *Design Automation and Test in Europe (DATE) Conference* (1999).

Chapter 5

VERIFICATION OF CONTROL PATHS USING PETRI NETS

Torsten Schober, Andreas Reinsch, and Werner Erhard
Jena University, Department of Computer Science, Ernst-Abbe-Platz 1-4, 07743 Jena, Germany; e-mail: schober@informatik.uni-jena.de, reinsch@informatik.uni-jena.de, erhard@informatik.uni-jena.de

Abstract: This work introduces a hardware design methodology based on Petri nets that is applied to the verification of digital control paths. The main purpose is to design control paths that are modeled and verified formally by means of Petri net techniques.

Key words: Petri nets; digital system design; hardware design methodology; property checking; verification and validation.

1. INTRODUCTION

As the complexity of digital systems grows rapidly, there is a rising interest to apply modeling and verification techniques for discrete-event systems. Owing to their universality, Petri nets can be applied for system modeling, simulation, and functional verification of hardware. Regarding digital system modeling and verification, the Petri net theory shows significant advantages:

- Inherent ability to model sequential and concurrent events with behaviors conflict, join, fork, and synchronization
- Ability to model system structure and system functionality as static and dynamic net behavior
- Ability to model digital systems at different levels of abstraction using hierarchical net concepts
- Ability to apply Petri nets as a graphical tool for functional and timed system simulation
- Ability to verify system properties by analysis of static and dynamic net properties

Considering the utilization of these advantages, a Petri net based hardware design methodology is introduced. The approach ends up in a complete and practicable hardware design flow. In order to achieve practical relevance and applicability, there are two requirements to accomplish:

1. Use of a common hardware description language for system modeling in addition to Petri nets, and
2. Use of a common logic synthesis tool to enable a technology mapping to hardware, where hardware platforms may range from field programmable gate arrays to custom integrated circuits.

2. PETRI NET BASED HARDWARE DESIGN METHODOLOGY

At the beginning of the system modeling process, it is essential to make two basic choices regarding net specification and net interpretation. The choice of a net specification determines the range of structural elements that can be represented by the system model. Thus, the chosen net specification has a great influence on the expressiveness of the system model. By means of place/transition nets with free-choice structure (FCPN), it is easy to express both sequential and concurrent events, as well as conflicting, joining, forking, and synchronizing events. Concerning analysis methods for structural and behavioral net analysis, FCPNs are particularly suitable too. Because of net interpretation, the components of a Petri net are assigned to the components of a digital system. Hence, every net interpretation of a Petri net creates a Petri net model. There exist several net interpretations in different technical domains that map either Petri net components to elementary hardware components or small subnets to hardware modules[1,2,3]. Control engineering interpreted (CEI) Petri nets[4] perform a direct mapping between a system of communicating finite state machines and a dedicated hardware realization. A CEI Petri net N is defined as a 10-tuple $N = (P, T, F_{\mathrm{pre}}, F_{\mathrm{post}}, m_0, G, X, Y, Q_{\mathrm{t}}, Q_{\mathrm{p}})$, namely places, transitions, pre- and post-arcs, an initial marking, transition guards, input signals, output signals, a transition-enabling function, and a place-marking function. Every transition t_{i} is assigned to an AND gate and every place p_{i} is assigned to a memory cell. Consequently, a marked place $p_{\mathrm{i}} = (e_{\mathrm{i}}, a_{\mathrm{i}})$ symbolizes an active memory cell in a state memory module. Transition activating and place marking are realized by two mappings Q_{t} and Q_{p}, where $Q_{\mathrm{t}} = G_{\mathrm{i}} * a_{\mathrm{i}}$ and $Q_{\mathrm{p}} = a_{\mathrm{i}}$. Therefore, in addition to the usual firing rule, transitions are activated by guards G_{i}, where $G_{\mathrm{i}} = f(x_{\mathrm{i}})$. These guards arise from logically conjuncted input signals x_{i} and express signal processing in a digital system. Output signals a_{i} can be used to enable state transitions $p_{\mathrm{i}} \to p_{\mathrm{j}}$ and for output signal processing y_{i}. In case of a state transition, $a_{\mathrm{i}} = 1$ is conjuncted with an arbitrary G_{j} at transition t_{j}.

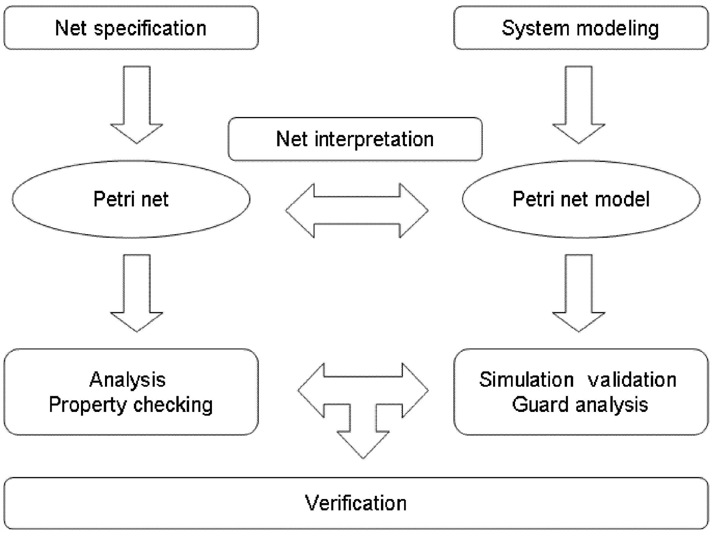


Figure 5-1. Petri net based modeling and verification of digital systems.

Then p_i is unmarked before p_j is marked, and hence a new state is activated. Concurrent behavior is represented by different state memory modules, each of sequential behavior, communicating through shared transitions. In that case t_j has several successors and/or predecessors p_j. State transitions in a CEI Petri net show that the token flow of the Petri net equivalently represents the signal flow of the modeled digital system. Digital system modeling and verification using interpreted Petri nets can be schematized as in Fig. 5-1.

Using CEI Petri nets and a place/transition net specification with FCPN, it is possible to model digital systems in a highly transparent way, such that system behavior is equivalently reproduced by the token flow of a simple structured Petri net model. The structure and behavior of a Petri net model is extensively analyzable if the model is retransformed into a FCPN. Therefore, all transition guards G_i are eliminated. To preserve state space equivalence it is required to apply a firing rule that only fires one transition at a time. For functional verification of a modeled digital system, a set of properties is studied by means of Petri net analysis. This approach is extended to describe a complete hardware design flow. A design flow shown in Fig. 5-2 is subdivided into five steps: modeling, verification, technology-independent net model mapping, logic synthesis, and timing verification.

Because of the high acceptance of conventional EDA synthesis tools and HDLs, it is not preferable to create a design flow that is solely based on Petri nets. Therefore, in the design flow two entities enable a transformation between constructs of a HDL and Petri net components, and vice versa. To translate VHDL constructs into Petri net structures, common rules[5] are used. On the top in

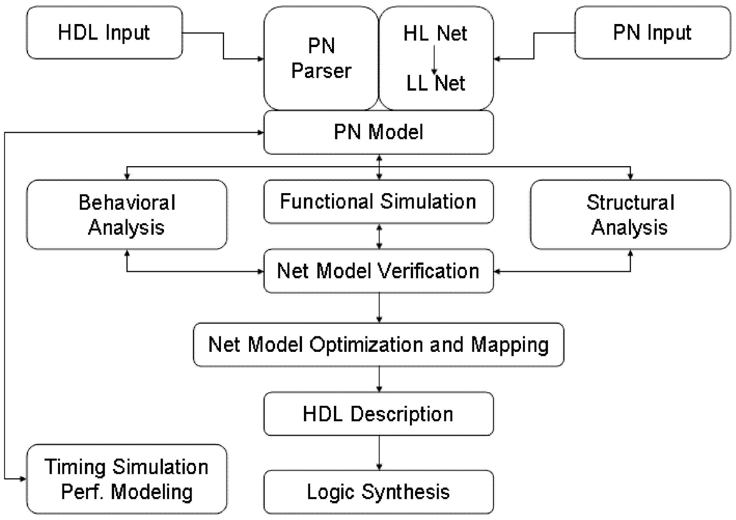


Figure 5-2. PN-based hardware design flow.

Fig. 5-2, HDL input as well as graphical or textual Petri net input are proposed. Regarding recent attempts to develop a general textual interchange format[6] for Petri nets to link Petri net editors, simulation tools, and analysis tools, an entity for textual Petri net input is very valuable. High-level Petri nets (HLPN), such as hierarchical Petri nets or colored Petri nets, can be applied if it is possible to unfold the HLPN to a FCPN. Once a Petri net model is created, simulation and analysis methods can be applied to verify the Petri net model functionally. Because of the graphical concepts of Petri nets, design errors can be detected easily by functional simulation. Also, functional simulation can be used to test subnets of the Petri net model successively. A simulation run is represented as a sequence of state transitions, and thus as a Petri net. The created occurrence sequence is analyzed further to evaluate system behavior. Beyond simulation, an exhaustive analysis of structural and behavioral properties leads to a formal design verification. Behavioral properties are studied by invariant analysis and reachability analysis. In a next step, the Petri net model can be optimized using advanced state space analysis, Petri net reduction techniques, and net symmetries. Therefore, some conditions should be met before a Petri net model can be optimized. Net model optimization techniques heavily effect an efficient implementation of the Petri net model[7]. The Petri net model is then subdivided into small subnets that can be mapped to dedicated hardware modules. The timing behavior of the chosen hardware modules determines the timing behavior of the designed digital system. Therefore, it is possible to verify a modeled digital system functionally and then implement it as a self-timed design[8] or as a synchronous clocked design. This is clearly not the case in a conventional hardware design methodology. A functionally verified and optimized Petri net model is transformed to dedicated VHDL constructs to enable logic synthesis by means of conventional EDA tools. After logic synthesis and technology-dependent design steps, simulation and analysis methods can be applied again to simulate and verify the timing behavior of the implemented design. Therefore, timing information is assigned to Petri net components of the Petri net model using a timed Petri net specification. For worst-case timing analysis deterministic timed Petri nets are suitable, while for performance modeling stochastic timed Petri nets are applied. It is shown that the complete design flow is based on Petri nets but, nevertheless, is embedded in a conventional design flow. To put the new design steps into practice, the tool development environment Petri Net Kernel (PNK)[9] and the Integrated Net Analyzer (INA)[10] are used to create the tool VeriCon[11], which performs the first two steps of this integrated hardware design flow in a prototypic way. The application of the proposed methodology is focused on microprocessor control paths.

3. VERIFICATION OF PETRI NET MODELS

For functional verification of a modeled digital system, a set of properties is studied by means of Petri net analysis. The analysis results are interpreted as properties of the Petri net model. In a more practical way, it is necessary to propose some requirements and goals that must be obtained for functional verification. For net models of microprocessor control paths, sets of analysis strategies are derived. These analysis strategies enable detection and localization of modeling errors. Each set of strategies is arranged to a scheme that enables an automated verification process. As an outcome of applied analysis strategies, some modeling guidelines are derived. Adhering to these guidelines enables to approach a functionally correct design already at early modeling cycles. Requirements for functional verification are expressed as Petri net properties of the analyzed Petri net and should cover

1. boundedness,
2. liveness,
3. reversibility,
4. state machine structure or free-choice structure,
5. persistence, and
6. pureness or proper self-loop assignment.

Petri net analysis has to ensure boundedness. In the Petri net model, *boundedness* determines that the modeled design has a finite state space. Consequently, control paths with an infinite number of states are not close to reality. If every place of the Petri net contains at most one token, logical values “high” and "low" are distinguishable for the memory cells that are represented by places. The digital system is thereby modeled transparently. Other assignment schemes require decoding. *Liveness* is the next necessary Petri net property for functional verification, and it is interpreted as the capability to perform a state transition in any case. Every transition of the Petri net can be fired at any reachable marking. Therefore, if liveness is preserved, the Petri net and thus the Petri net model are not deadlocked. If, in a live Petri net, the initial state is reachable, then the Petri net is reversible. To reflect the structure of a sequential control path, the Petri net should have state machine structure (SM structure). In this case no transition of the Petri net is shared. Every transition has exactly one predecessor place and one successor place. Therefore, in a state machine, generation and consumption of tokens is avoided. For modeling concurrent control paths, both marked graph structures (MG structure) with shared transitions, forking, and synchronizing events and SM structures are required. Thus the Petri net model of a concurrent control path should have free-choice structure. Places with more than one successor transition generate conflict situations. If several posttransitions of a marked place are enabled, and one transition fires, then all other transitions are disabled. Hence the Petri net is not persistent, and also it is not predictable as to what transition will fire. Transition guards are able to solve conflicts, because they represent an additional firing condition that is required to perform a state transition. Therefore, transitions in conflict can become unique using transition guards and no behavior ambiguity remains. When a pre-place of a transition also appears as its post-place, then there is a self loop in the Petri net. Self loops can give a structural expression to model external signal processing distinctly. It has to be clarified in the modeling process as to which, if any, which self loops are desired to emphasize external signal processing. Thus, self loops can be detected and assigned to that situation, and others are marked as modeling errors and should be removed. As for concurrent control path models, it is not preferred to start with a state space analysis. Because of the huge amount of states that a concurrent control path may have, it is more convenient to apply structural properties analysis first. Tests for marking conservation, coverability with place invariants, and strong connectedness perform fast. Out of these net model properties, boundedness is decided.

4. ANALYSIS STRATEGIES

Table 5-1 summarizes all derived analyses strategies[12] regarding detected modeling errors and affected Petri net properties. Strategies S1 ... S9 are applied to verify sequential control paths. Primed strategies are derived from nonprimed ones with only minor changes and can be applied for pipelined control path verification. If a strategy (S1 ... S9) exists only as nonprimed version, it can

Table 5-1. Analysis strategies

Analysis strategy	Modeling error	Affected property
S1, S1′, S1″	Transition without a pre-place, token production, fork	Boundedness
S2	Not strongly connected	Liveness
S3	Token consumption	Liveness, SM structure
S4, S4′	Not conservative	SM structure, conservativeness
S5	Transition without post-place	Liveness, SM structure
S6	Place without pretransition	Liveness
S7, S7′	Place without posttransition, and with nonconservative pretransitions	Liveness, boundedness
S8, S8′	Self loops	Pureness
S9, S9′	Conflicts	Persistence
S10	Marking sum > 1	Safeness
S11	Nonsynchronized transitions between pipeline stages	Safeness, liveness
S12	Marking sum < 1	Liveness

be applied for both types of control paths. Strategies S10 … S12 are derived to verify pipelined control paths.

An automated verification of Petri net models using S1 … S12 can be performed according to Figures 5-3 and 5-4. In Fig. 5-3, verification of sequential control paths is shown, whereas Fig. 5-4 illustrates pipelined control path verification. Using PNK and INA, all analysis strategies are efficiently implementable. Exemplarily, strategies S1 and S2 are listed.

4.1 Strategy S1

1. If Petri net N is unbounded, then
   (a) determine all shared transitions t_i, $|t_i \bullet| > 1$ using $\bullet p_j \,\forall\, p_j \in P \Rightarrow$ generate list $\{t_{iP}\}$.
   (b) determine all transition sources Ft_i0 using $\bullet t_i \,\forall\, t_i \in T$.
2. Check if $[(t_i = \bullet p_j,\, t_i \in \{t_{iP}\}) \vee (t = Ft_i0)] \,\forall\, p_j \in P$. $\Rightarrow$ pretransition of place p_j produces tokens $t_i \in \{t_{iP}\}$ or Ft_i0, and unboundedness of p_j is caused by t_i.

4.2 Strategy S2

1. Check liveness $\Rightarrow$ generate list of live transitions $\{t_{iL}\}$.
2. Compute strongly connected components (SCC) in $R_N \Rightarrow$ generate tuple $\{SCC_i, p_j\}$.

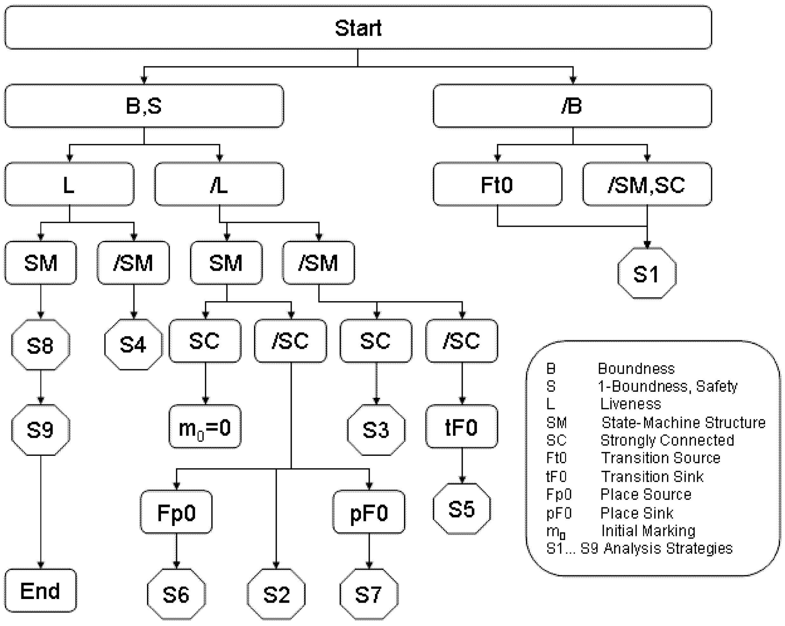


Figure 5-3. Application of analysis strategies for sequential control paths.

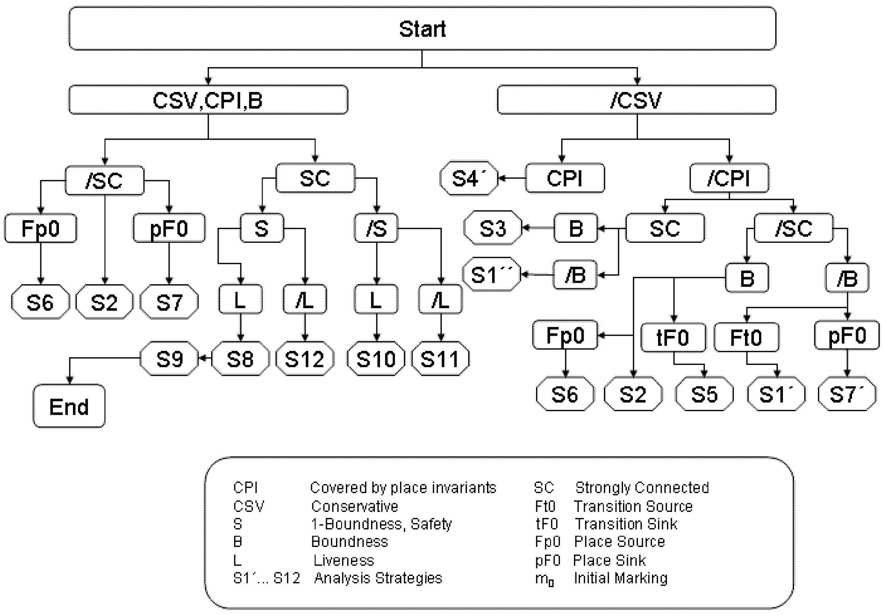


Figure 5-4. Application of analysis strategies for pipelined control paths.

3. Determine shared places and sharing transitions for each SCC using preset and postset of $p_j \forall p_j \in P$: $(|\bullet p_j| > 1) \vee (|p_j \bullet| > 1) \Rightarrow$ generate tuple $\{SCC_i, p_j, t_k\}$.
4. Compare all tuples $\{SCC_i, p_j, t_k\}$. Multiple-occurring t_k form transitions between different SCCs: $SCC_i \rightarrow SCC_j$ ($SM_i \rightarrow SM_j$).
5. Compare all t_k and $\{t_{iL}\}$. If $t_k \in \{t_{iL}\} \Rightarrow t_k$ is live. Otherwise t_k is a dead transition between two different SCCs.
6. Detect live SCCs. For each SCC, check whether $(\bullet p_j \in \{t_{iL}\} \forall p_j \in SCC_i) \vee (\bullet p_j \in \{t_{iL}\} \forall p_j \in SCC_i) \Rightarrow SCC_i$ is live. Dead SCCs include at least one dead transition in the preset or postset of its places.

Strategies S1 … S1″ detect shared transitions with more than one post-place or transitions without pre-place that cause unbounded places, and hence an unbounded Petri net. S2 and S3 are applied to detect and localize dead transitions that are caused by lack of strong connectedness or by shared transitions with more than one pre-place. It is possible that the analyzed Petri net is 1-bounded and live, but it shows no state machine structure. In this case, all generated tokens are consumed within a marked graph that is a Petri net structure in which no place is shared. Strategy S4 is applied to localize such shared transitions. Similarly S4′ finds nonconservative transitions by evaluating place invariants. By means of strategies S5 … S7′, dead transitions caused by one-sided nodes are detected, as mentioned in Table 5-1. In analysis strategies S8 … S9′, transition guards are used to interpret conflicts and self loops within the Petri net model. The last strategies S10 … S12 are applied to localize liveness and safeness problems in conservative, strongly connected, and place invariant covered net models. According to the proposed analysis strategies, it is possible to derive modeling guidelines that affect the modeling process and assist the designer to create a system model reflecting the desired functionality. In Table 5-2, analysis results and their interpretation for the Petri net model are summarized.

Table 5-2. Verified properties and their interpretation in the Petri net model

Petri net properties	Interpretation in the Petri net model
State machine structure	Sequential control path
Free-choice structure	Concurrent control path
Boundness	Signal to token assignment
Liveness	No deadlocks, no restrictions of state transitions
Strong connected	Arbitrary state transitions
No source and sink	No restrictions of liveness and/or boundedness
Reversibility	Resetable control path
Pureness	No self loops
Persistence	Solved dynamic conflicts

4.3 Modeling guidelines

- Avoid nonconservative transitions
- Avoid of one-sided nodes
- Ensure strong connectedness
- Remove conflicts using transition guards
  1. Provide all posttransitions of a shared place with guards
  2. Provide every posttransition of a shared place with a unique guard

5. APPLICATION

In computer architecture, the DLX microprocessor is a well-known example[13]. In a case study, the control path of the sequential and the concurrent DLX is designed as a Petri net model, whereas data path and memory test bench are provided by VHDL models[14]. In all, 52 instructions covering all instruction types are modeled. Additionally reset, error, exception, and interrupt handling is considered. The Petri net model corresponding to the sequential control path contains 243 nodes. To implement the whole instruction set, 64 control states are required. Complex sequential control paths, such as control path of a sequential microprocessor, consist of a system of strongly connected state machines. This includes decomposability into partial nets with state machine structure. When a Petri net is 1-bounded and live and has state machine structure, then a very transparent Petri net model is created. Every state in the Petri net model is assigned to a state of the control path. Thus, the reachability tree is rather small and represents exactly 64 control states. Using the derived analysis strategies S1 . . . S9, all modeling errors could be detected, localized, and removed. The analyzed Petri net properties and their interpretation for the Petri net model enable to decide functional correctness of the Petri net model formally. The Petri net model that corresponds to the pipelined control path contains 267 nodes and, compared with the sequential case, a similar proportion of places and transitions. But the state space of this Petri net model is of $O(10^5)$. The processor pipeline with five stages is modeled using a free-choice structure that shows only conservative transitions. Hence the fork degree of a transition equals its synchronization degree. There are control places to ensure mutual exclusion between pipeline stages. Under these modeling conditions, especially, place invariants are convenient to verify the correct pipeline behavior. A Petri net model of 5 pipeline stages leads to 4 place invariants that cover the whole net model, each a set of places that contains a constant weighted sum of markings. If there are less than 4 invariants, then there may occur a situation of an unbounded net model. If the weighted sum of markings is not equal to 1, then there is a liveness or safeness problem. Because of structural preanalysis, it is easy to detect and locate modeling errors. The analysis strategies $S1' \dots S12$ detected and localized all occurring modeling errors. Thus the Petri net model of the pipelined control path was formally verified, too.

6. CONCLUSIONS

This work introduces a Petri net based hardware design methodology for modeling and verification of digital systems. Modeling digital systems using free-choice Petri nets (FCPN) and control engineering interpreted Petri nets (CEI PN) leads to highly transparent and simple structured Petri net models. Using Petri net analysis techniques, functional verification of Petri net models is obtained by analysis of Petri net properties and a suitable interpretation of the Petri net model. For functional verification of control paths, analysis strategies are provided. Using these analysis strategies, it is possible to detect and localize modeling errors automatically. As an outcome of applied analysis strategies, some modeling guidelines are derived. Adhering to these modeling guidelines enables to approach a functionally correct design already at early modeling cycles. The methodology is applied to the functional verification of microprocessor control paths.

REFERENCES

1. S. Patil, Coordination of asynchronous events. PhD Thesis, Department of Electrical Engineering, MIT, Cambridge, MA (1970).
2. D. Misunas, Petri-nets and speed-independent design. *Communications of the ACM*, **16** (8), 474–479 (1973).
3. J. Cortadella et al., Hardware and Petri nets: Application to asynchronous circuit design. *Lecture Notes in Computer Science*, **1825**, 1–15, (Springer 2000).
4. R. König, L. Quäck, *Petri-Netze in der Steuerungstechnik*. Oldenbourg, München (1988).
5. S. Olcoz, J.M. Colom, A Petri net approach for the analysis of VHDL descriptions. *Lecture Notes in Computer Science*, **683**, 1–15 (Springer 1993).
6. M. Jüngel, E. Kindler, M.Weber, The Petri net markup language. In: Philippi Stefan (ed.), *Fachberichte Informatik Universität Koblenz-Landau,* Nr. 7-2000, pp. 47–52 (2000).
7. W. Erhard, A. Reinsch, T. Schober, Formale Verifikation sequentieller Kontrollpfade mit Petrinetzen, *Berichte zur Rechnerarchitektur, Universität Jena, Institut für Informatik*, **7** (2), 1–42 (2001).
8. W. Erhard, A. Reinsch, T. Schober, First steps towards a reconfigurable asynchronous system. In: *Proceedings of 10th IEEE International Workshop on Rapid System Prototyping (RSP)*, Clearwater, FL, pp. 28–31 (1999).
9. E. Kindler, M. Weber, The Petri net kernel—An infrastructure for buildung Petri net tools. *Lecture Notes in Computer Science*, **1643**, 10–19 (Springer 1999).
10. INA Integrated Net Analyzer. Humboldt Universität zu Berlin, Institut für Informatik, (July 7, 2003); http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina.

11. K.Wagner, Petri Netz basierte Implementierung einer formalen Verifikation sequentieller Kontrollpfade. Studienarbeit, Universität Jena, Institut für Informatik, pp. 1–22 (2002) (unpublished).
12. T. Schober, Formale Verifikation digitaler Systeme mit Petrinetzen. Dissertation, Universität Jena, pp. 1–114 (2003).
13. J.A. Hennessy, D.A. Patterson, *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publisher, San Francisco (1996).
14. The DLXS processor design, University Stuttgart, Institut of Parallel and Distributed Systems (April 28, 1998); http://www.informatik.uni-stuttgart.de/ipvr/ise/projekte/dlx/.

Chapter 6

MEMORY-SAVING ANALYSIS OF PETRI NETS

Andrei Karatkevich
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgórna 50, 65-246 Zielona Góra, Poland; e-mail: A.Karatkevich@iie.uz.zgora.pl

Abstract: An approach to Petri net analysis by state space construction is presented in the paper, allowing reducing the necessary memory amount by means of removing from memory the information on some of intermediate states. Applicability of the approach to deadlock detection and some other analysis tasks is studied. Besides this, a method of breaking cycles in oriented graphs is described.

Key words: Petri nets; simulation; analysis; formal models.

1. INTRODUCTION

Petri nets[1] are a popular formal model of a concurrent discrete system, widely applied for specifying and verifying control systems, communication protocols, digital devices, and so on. Analysis of such nets is a time- and memory-consuming task, because even a simple net may have a huge number of reachable states caused by its concurrent nature (the so-called state explosion problem[2,3]).

However, state space search remains one of the main approaches to Petri net analysis (deadlock detection, for example). But there are various methods handling state explosion problem, such as *lazy state space constructions*, building reduced state spaces instead of the complete ones[2]. Among such methods, Valmari's *stubborn set method*[4,5] is best known.

Theoretically there is no necessity of huge memory amount to solve Petri net analysis problems; there is a polynomial-space algorithm of deadlock detection in a safe Petri net, but it is practically absolutely inapplicable because its time consumption is woeful[3]. Generally, the known algorithms solving verification tasks in a relatively small memory are extremely slow[3].

In this paper we concentrate on a less radical approach, reducing memory amount and keeping it, however, exponential in worst case. The approach is based on removing from memory some of the intermediate states. Some results are recalled from Refs. 6 and 7; also, new results are presented.

2. PRELIMINARIES

A *Petri net*[1] is a triple $\Sigma = (P, T, F)$, where P is a set of *places*, T is a set of *transitions*, $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$. For $t \in T$, ${}^\bullet t$ denotes $\{p \in P | (p, t) \in F\}$, $t^\bullet$ denotes $\{p \in P | (t, p) \in F\}$, and ${}^\bullet t$ and $t^\bullet$ are the *sets of input* and *output places*, respectively. $\forall t \in T : {}^\bullet t \neq \emptyset, t^\bullet \neq \emptyset$. A similar notation is used for places (${}^\bullet p$, $p^\bullet$). A Petri net can also be considered as an oriented bipartite graph. A *state* (*marking*) of a net is defined as a function $M\colon P \to \{0, 1, 2, \ldots\}$. It can be considered as a number of *tokens* situated in the net places. $M(p)$ denotes the number of tokens in place p at M. $M' > M$ denotes that $\forall p \in P\colon M'(p) \geq M(p)$ and $\exists p \in P\colon M'(p) > M(p)$. *Initial state* M_0 is usually specified.

A transition t is *enabled* and can *fire* if all its input places contain tokens. Transition firing removes one token from each input place and adds one token to each output place, thus changing the current state. If t is enabled in M and its firing transforms M into M', then that is denoted as MtM'. This denotation and the notion of transition firing can be generalized for *firing sequences*s (sequential firing of the transitions, such that each transition is enabled in the state created by firing of the previous transition). If a firing sequence σ leads from state M to M', it is denoted as $M\sigma M'$. A state that can be reached from M by a firing sequence is called *reachable* from M; the set of reachable states is denoted as $[M\rangle$. A transition is *live* if there is a reachable marking in which it is enabled; otherwise it is *dead*. A state in which no transitions are enabled is called a *deadlock*. A net is *live* if in all the reachable markings, all the transitions of the net are live. A net is *safe* if in any reachable marking no place contains more than one token. A net is *bounded* if $\exists n\colon \forall p \in P\ \forall M \in [M_0\rangle M(p) \leq n$ (there is an upper bound of number of tokens for all the net places in all reachable markings).

A *reachability graph* is a graph $G = (V, E)$ representing state space of a net. $V = [M_0\rangle; e = (M, M') \in E \Leftrightarrow MtM'$ (then t marks e). The reachability graph is finite if and only if the net is bounded. A *strongly connected component* (SCC) of a reachability graph is a maximal strongly connected subgraph. A *terminal component* of a graph G is its SCC such that each edge which starts in the component also ends in it[3,5].

A set T_S of the transitions of a Petri net at state M is a *stubborn set* if (1) every disabled transition in T_S has an empty input place p such that all transitions in ${}^{\bullet}p$ are in T_S, (2) no enabled transition in T_S has a common input place with *any* transition (including disabled ones) outside T_S, and (3) T_S contains an enabled transition.

A *reduced reachability graph* (RRG), created with the basic *stubborn set method*, is a subgraph of the reachability graph built so that in every considered state only firing of the enabled transitions belonging to T_S is simulated. Such an RRG contains all deadlocks of the system that are reachable from the initial states. Furthermore, all deadlock states of the RRG are deadlock states of the system (the fundamental property of the stubborn set method[3,4]).

3. ON-THE-FLY REDUCTION OF REACHABILITY GRAPH

Consider the problem of detecting deadlocks. To solve it there is no necessity to keep in memory the whole (even reduced) reachability graph with all intermediate (non-deadlock) states. But it is evident that removing all of them can lead to eternal looping (if the reachability graph has cycles). So, some of the intermediate states should be kept. Which ones? The following affirmations allow obtaining the answer.

Affirmation 1. (Lemma 3 from Ref. 7). For every cycle C in a reachability graph, there is a cycle C_Σ in the net graph such that every transition belonging to C_Σ marks an arc in C.

Affirmation 2. (Lemma 1 from Ref. 7). Let $M\,\sigma\,M'$, where $M' > M$. Then there is a cycle C_Σ in the net graph such that every transition belonging to C_Σ appears in σ.

An algorithm is presented, which is a modification of the well-known algorithm of reachability graph building.

Algorithm 1

```
Input: Petri net $\Sigma = (P, T, F)$, initial state $M_0$.
Output: Graph $G(V, E)$.
1 $V := \{M_0\},\ E := \emptyset - with - circle,\ D := \{M_0\}$. Tag $M_0$ as "new."
2 Select $Q \subseteq T$ such that for every cycle in the net graph, at least one
  transition belongs to $Q$.
3 While "new" states exist in $V$, do the following:
3.1     Select a new state $M$.
3.2     If no transitions are enabled at $M$, tag $M$ as "deadlock."
3.3     While there exist enabled transitions at $M$, do the following for each
  enabled transition $t$ at $M^*$:
3.3.1   Obtain the state $M'$ that results from firing $t$ at $M$.
```

3.3.2 If on the path from M_0 to M, there exists a marking M'' such that $M > M''$, then communicate "The net is unbounded" and go to 4.
3.3.3 If $M' \notin V$, add M to V and tag M' as "new."
3.3.4 Add the arc (M, M') to F.
3.3.5 If $t \in Q$, add M' to D.
3.4 If M is not a "deadlock" and $M \notin Q$, do the following:
3.4.1 For every pair of arcs in $(a, M) \in F$ and $(M, b) \in F$, add to F arc (a, b).
3.4.2 Remove M from V and all the incident arcs from F.
4 The end.

Affirmation 3. Algorithms 1 and 1a stop for every Petri net.
Proof. Suppose the net is bounded and the algorithm never stops. This means that there exists a cycle in the reachability graph such that every state in it is added to V and then removed from it, and the loop never stops. But from Affirmation 1 it follows that at least one of those states will be included in set D and never deleted from V; hence it cannot be tagged as "new" more than once, and such eternal looping is impossible. A contradiction.

Suppose the net is unbounded and the algorithm never stops. Then the algorithm considers new states (never considered before) infinitely often (another possibility of looping is excluded by Affirmation 1). Any "long enough" firing sequence leading to new states will go through states M and M' such that $M' > M$, which follows from the fact that P is finite. Then, as follows from Affirmation 2, a state M'' between them will be added to D and never removed. This means that graph G will grow infinitely.

This in turn means that sooner or later in G there will be a "long enough" path such that there will be states M and M' in it such that $M' > M$. Then the algorithm will detect this situation (item 3.3.2) and stop.

The proof is now completed; note that it is valid for both variants of the algorithm (proof only for Algorithm 1 would be simpler).

Affirmation 4. Algorithms 1 and 1a detect all the deadlocks of a Petri net or its unboundedness.

The proof follows from the algorithm description and the fundamental property of the stubborn set method.

Note that item 2 of Algorithm 1 is a nontrivial task if we want to make set Q as small as possible (which would reduce the needed memory amount). It can be formulated as a task of *minimal decyclization* of an oriented graph or of finding *minimal feedback arc set*[9]. It has applications in electrical engineering, computer-aided design of discrete devices, scheduling, and so on. The topic is discussed in the appendix.

*As Algorithm 1a the variant of Algorithm 1 will be meant, in which item 3.3 is the following: "While there exist enabled transitions belonging to T_S at M, do the following for each enabled transition $t \in T_S$:"

4. SOLVING ANALYSIS TASKS

The application of the proposed approach to solving some other analysis tasks will be discussed below.

- **Boundedness**. It is easy to see that n-boundedness can be checked by Algorithm 1 if the number of tokens in the places is checked on-the-fly. Algorithm 1 also detects unboundedness, as follows from Affirmations 2 and 3.
- **Liveness**. It is easy to see that Algorithm 1—as it is—cannot check liveness. Consider the next modification (the additions are given in bold).

Algorithm 2

```
Input: Petri net $\Sigma = (P, T, F)$, initial state $M_0$.
Output: Graph $G(V, E)$.
1   $V := \{M_0\}$, $E := \emptyset$, $D := \{M_0\}$. Tag $M_0$ as “new.”
2   Select $Q \subseteq T$ such that for every cycle in the net graph, at least one
    transition belongs to $Q$.
3   While “new” states exist in $V$, do the following:
3.1   Select a new state $M$.
3.2   If no transitions are enabled at $M$, tag $M$ as “deadlock.”
3.3   While there exist enabled transitions at $M$, do the following for each
      enabled transition $t$ at $M$:
3.3.1   Obtain the state $M'$ that results from firing $t$ at $M$.
3.3.2   If on the path from $M_0$ to $M$ there exists a marking $M''$ such that
        $M > M''$, then communicate “The net is unbounded” and go to 4.
3.3.3   If $M' \notin V$, add $M$ to $V$ and tag $M'$ as “new.”
3.3.4   Add the arc $(M, M')$ to $F$, **mark $(M, M')$ by $t$**.
3.3.5   If $t \in Q$, add $M'$ to $D$.
3.4   If $M$ is not a “deadlock” and $M \notin Q$, do the following:
3.4.1   For every pair of arcs in $(a, M) \in F$ and $(M, b) \in F$, add to $F$ arc $(a, b)$,
        and **mark $(a, b)$ by all the transitions marking $(a, M)$ and $(M, b)$**.
3.4.2   Remove $M$ from $V$ and all the incident arcs from $F$.
4   The end.
```

The graph built by Algorithm 2 contains enough information for liveness checking—it is enough to check whether every transition marks an arc in every terminal component.

- **Reversibility**. The net is reversible if and only if graph G constructed by Algorithm 1 is strongly connected.
- **Reachability**. Algorithm 1 by construction considers every reachable state, and so, of course, reachability of a state can be checked by means of it.

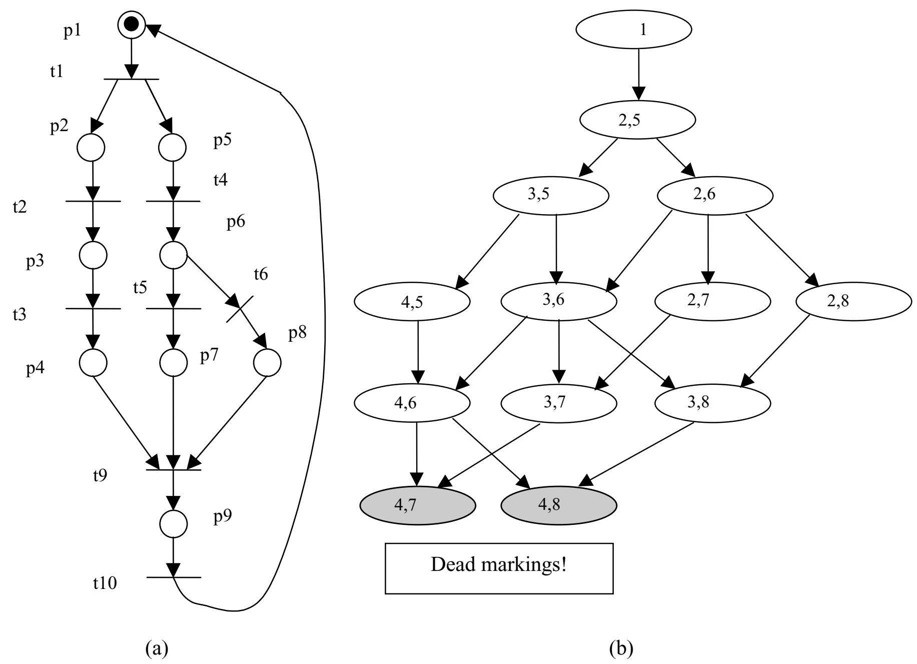


Figure 6-1. A Petri net (a) and its reachability graph (b).

5. EXAMPLE

Consider a Petri net (Fig. 6-1a). Its reachability graph is shown in Fig. 6-1b. It has 13 nodes. In Fig. 6-2 graph G is shown; Fig. 6-2a presents it at a stage of Algorithm 1 execution when its number of nodes is maximal (supposing search in BFS order); Fig. 6-2b presents its final form. The maximal number of nodes of G is 6; every marking has been considered only once. The situation is different if the state space is searched in DFS order; then the maximal number of nodes is also 13, but some of the states have been considered two or even three times.

It is also interesting to compare an RRG built using the stubborn set method and graph G' built by Algorithm 1a for this example. The RRG contains seven nodes, and G' has maximally three nodes.

6. CONCLUDING REMARKS

6.1 Complexity of the method

How much we gain (and loose) with the proposed method?

An exact analytical evaluation of the space and time complexity of the method turns to be a complex task even for the nets with a very restricted

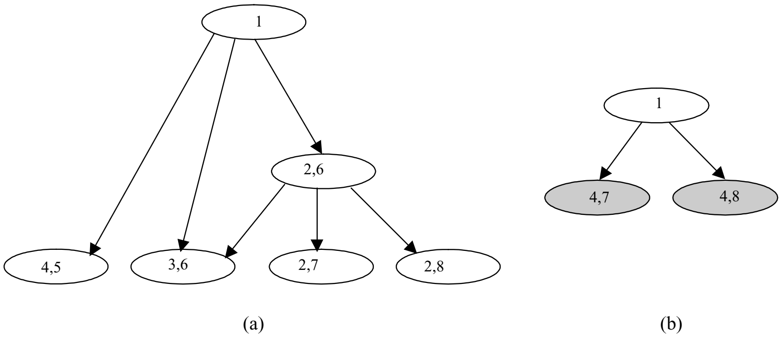


Figure 6-2. Intermediate (a) and final (b) graph *G* constructed by Algorithm 1 for the net shown in Fig. 1a.

structure, and we do not have sufficient evidence to decide how good it is generally. There are a lot of "good" and "bad" examples. Some general notes on its complexity are presented below.

- Gaining in memory, we lose in time with the method. In most cases the algorithms described have to recalculate the same states several (sometimes a number of) times.
- BFS search seems to be preferable here. DFS in many cases allows saving more space, but it leads to multiple recalculation of the states and increases time consumption drastically.
- A combination of the method with the stubborn set method is efficient, because the stubborn set method, avoiding interleaving, reduces radically the number of paths leading to a node in a reachability graph. So, recalculation of the states occurs rarely for such a combination.

6.2 Applicability of the method and further work

The approach presented allows saving memory when solving the tasks of Petri net analysis by methods of state space search. Space complexity is reduced at the expense of time complexity, so the practical use of the approach makes sense (if any) in the cases where the amount of memory is a more critical parameter than time. The approach seems to be especially efficient for deadlock and reachability analysis and, generally, for safety properties (to a wide extent)[4].

The method is compatible with the stubborn set method. It would be interesting to analyze how it will be working with other lazy state space construction methods; for example, with maximal concurrent simulation[9,10]. An experimental analysis of the method using benchmarks could answer the question whether it is good for practical purposes.

ACKNOWLEDGMENT

The research was supported by the Polish State Committee for Scientific Research (Komitet Badań Naukowych) grant 4T11C 006 24.

APPENDIX: BREAKING CYCLES IN ORIENTED GRAPHS

Some methods of optimal "decyclization" of oriented graphs are described in Refs. 11 and 12. One of them is based on the fact that the adjacency matrix of an acyclic oriented graph can be transformed (by reordering columns and rows) to a strictly triangular matrix. The method reorders the matrix in such a way that the number of nonzero elements below the main diagonal or on it is minimized. Then the arcs corresponding to such nonzero elements are removed. Of course such reordering is a complex combinatorial task.

Another method described in Ref. 12 is based on Boolean transformations: let a Boolean variable correspond to every arc of the graph, and elementary disjunction of those variables (without negation) correspond to every cycle in the graph. Transform the obtained formula in conjunctive normal form (CNF) into disjunctive normal form (DNF) (by multiplying the disjunctions and deleting products that subsume others) and select the shortest elementary conjunction, which will correspond to the minimal feedback arc set.

By using some newer methods of Boolean transformations, this approach can be much refined. In fact the shortest prime implicant has to be calculated here. A very efficient method of computation of the prime implicants of CNF is proposed by Thelen[13], where the prime implicants are obtained without direct multiplying but by using a search tree and the reducing rules for it. In Ref. 14, a modification of Thelen's method intended for computation of the shortest prime implicant is presented. In Refs. 15 and 16, the heuristics for Thelen's method are proposed (not affecting the results but additionally reducing the search tree). On the other hand, the task can be reduced to the deeply investigated unate covering problem[17]. Therefore, much can be done to accelerate the exact solving of the task.

But computing of all the cycles in a graph requires exponential time and space in the worst case; computation of the shortest prime implicant is NP, the same as that of the covering problem. For the problem being the topic of our paper, as well as for other practical purposes, a quick approximate algorithm would be useful. Such an algorithm is proposed in Ref. 8.

In this algorithm the weights are assigned to the arcs of the graph according to the formula:

$$w(e) = id(init(e)) - od(init(e)) + od(ter(e)) - id(ter(e)),$$

where e is an arc, $w(e)$ is the weight, *id* and *od* are input and output degrees, respectively, and *init*(e) and *ter*(e) are the initial and terminal nodes of the arc e, respectively. Then the arcs are sorted (in order of nondecreasing weights) and added to the acyclic oriented graph being constructed, excluding the arcs, adding of which would create a cycle. The algorithm processes each strongly connected component separately.

This process is similar to the process of building of a minimal spanning tree in Prim's algorithm[18], but, of course, a greedy algorithm cannot guarantee the optimal solution in this case. The intuition behind the algorithm is the following: if the initial node of an arc has many incoming arcs and few outgoing arcs, and its terminal node has many outgoing arcs and few incoming arcs, then it is likely that it belongs to many cycles and is one (or one of the few) common arc of those

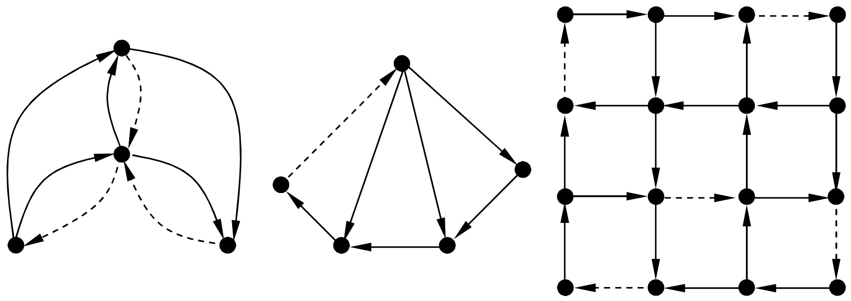


Figure 6-3. Examples of breaking cycles in oriented graphs by the algorithm described in the appendix. Dashed arcs are removed.

cycles. Therefore, it is better not to add such an arc to the acyclic graph being built; that is why a bigger weight is assigned to it.

The time complexity of the algorithm is $\Theta((|V| + |E|)^2)$. For many examples it gives exact or close to exact solutions (see Fig. 6-3).

REFERENCES

1. T. Murata, Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, **77** (4), 548–580 (April 1989).
2. M. Heiner, Petri net-based system analysis without state explosion. *High Performance Computing '98*, Boston, pp. 1–10 (April 1998).
3. A. Valmari, The state explosion problem. In: W. Reisicg, G. Rozenberg (eds.), *Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS* **1491**, Springer-Verlag pp. 429–528, (1998).
4. A. Valmari, State of the art report: Stubborn sets. *Petri Nets Newsletter*, **46**, 6–14 (1994).
5. C. Girault, R. Valk *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*, Springer-Verlag (2003).
6. A. Karatkevich, M. Adamski, Deadlock analysis of Petri nets: Minimization of memory amount. In: *Proceedings of 3rd ECS Conference*, Bratislava, pp. 69–72 (2001).
7. A.G. Karatkevich, Dynamic reduction of Petri net reachability graphs. *Radioelektronika i Informatika*, **18** (1), 76–82, Kharkov (2002) (in Russian).
8. A. Karatkevich, On algorithms for decyclisation of oriented graphs. In: *DESDes'01*. Przytok, Poland, pp. 35–40 (2001).
9. R. Janicki, M. Koutny, Optimal simulation, nets and reachability graphs. Technical Report No. 01-09, McMaster University, Hamilton (1991).
10. A. Karatkevich, Optimal simulation of α-nets. In: *SRE-2000*. Zielona Góra, pp. 205–210 (2000).
11. A. Lempel, Minimum feedback arc and vertex sets of a directed graph *IEEE Transactions on Circuit Theory*, **CT-13** (4), 399–403 (December 1966).
12. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, New Jersey (1974).
13. B. Thelen, *Investigation of Algorithms for Computer-Aided Logic Design of Digital Circuits*. University of Karlsruhe (1988) (in German).

14. H.-J. Mathony, Universal logic design algorithm and its application to the synthesis of two-level switching circuits. In: *Proceedings of the IEE*, **136** (3), 171–177 (1989).
15. J. Bieganowski, A. Karatkevich, Heuristics for Thelen's prime implicant method. In: *MSK*, Krakow, pp. 71–76 (November 2003) (in Polish).
16. A. Węgrzyn, A. Karatkevich, J. Bieganowski, Detection of deadlocks and traps in Petri nets by means of Thelen's prime implicant method, *AMCS*, **14** (1), 113–121 (2004).
17. O. Coudert, J.K. Madre, New ideas for solving covering problems. In: *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, San Francisco, California, United States, pp. 641–646 (1995).
18. T.H. Cormen, Ch.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*. MIT (1994).

Chapter 7

SYMBOLIC STATE EXPLORATION OF UML STATECHARTS FOR HARDWARE DESCRIPTION

Grzegorz Łabiak
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgórna 50, 65-246 Zielona Góra, Poland; e-mail: G.Labiak@iie.uz.zgora.pl

Abstract: The finite state machine (FSM) and Petri net theories have elaborated many techniques and algorithms that enable the employment of formal methods in the fields of synthesis, testing, and verification. Many of them are based on symbolic state exploration. This paper focuses on the algorithm of the symbolic state space exploration of controllers specified by means of statecharts. Statecharts are a new technique for specifying the behaviour of controllers, which, in comparison with FSM and Petri nets, is enriched with notions of hierarchy, history, and exception transitions. The paper presents the statechart diagrams as a means of digital circuit specification.

Key words: statechart; UML; logic control; symbolic analysis; BDD.

1. INTRODUCTION

Statecharts are a visual formalism for the specification of reactive systems, which is based on the idea of enriching state transition diagrams with notions of hierarchy, concurrency, and broadcast communication[1,2,3,4]. It was invented as a visual formalism for complex systems by David Harel[1]. Today, as a part of UML technology, it is widely used in many fields of modern engineering[5]. The presented approach features such characteristics as Moore and Mealy automata, history, and terminal states. There are many algorithms based on a state transition graph traversal for finite state machines (FSMs) which have applications in the area of synthesis, test, and verification[4,6,7,8,9]. It seems to be very promising to use well-developed techniques from the FSM and Petri net theory in the field of synthesis[10], testing, and the verification of controllers specified by means of statechart diagrams. These considerations caused the elaboration of the new algorithms of symbolic state space exploration.

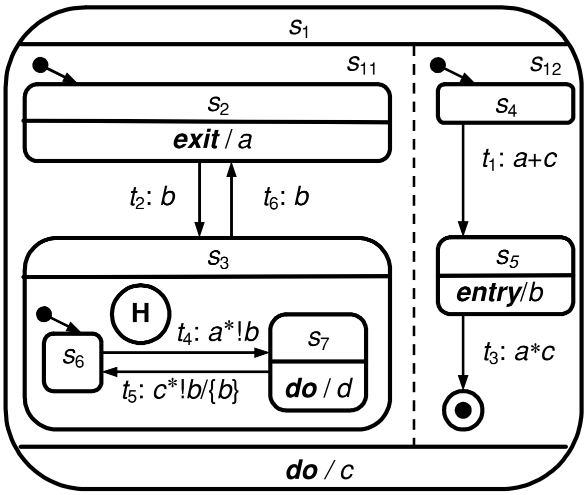
2. SYNTAX AND DEFINITIONS

The big problem with statecharts is syntax and semantics. A variety of application domains caused many authors to propose their own syntax and semantics[11], sometimes differing significantly. Syntax and semantics presented in this paper are intended for specifying the behavior of binary digital controllers that would satisfy as much as possible the UML standard[5]. Not every element of UML statechart syntax is supported in the HiCoS approach. The selection of language characteristic was made on the basis of application domain and technical constraints of programmable logic devices.

As a result of these considerations it was assumed that system HiCoS is to be intended for untimed control systems which operate on binary values. Moreover, the research was divided into two stages, delimiting them with a modular paradigm[12]. Hence, HiCoS statecharts are characterized by hierarchy and concurrency, simple states, composite states, end states, discrete events, actions assigned to state (*entry*, *do*, *exit*), simple transitions, history attribute, and logic predicates imposed on transitions. Another very essential issue is to allow the use of feedbacks: this means that events generated in circuits can affect their behavior. The role of an end state is to prevent removing an activity from a sequential automaton before the end state becomes active. Such elements as factored transition paths and time were rejected, whereas others as cross-level and composite transitions and synch states have been shifted to the second stage of the research. An example of statechart is depicted in Fig. 7-1, where event *a* is an input to the system and event *b* is an output. Events *b* and *c* are of local scope. Figure 7-2 depicts the hierarchy tree (a.k.a. AND-OR tree).

3. SEMANTICS

A digital controller specified with a statechart and realized as an electronic circuit is meant to work in an environment that prompts the controller by means of events. It is assumed that every event (incoming, outgoing, and internal) is bound with a discrete time domain. The controller reacts to the set of accessible events in the system through a firing set of enabled transitions called a *microstep*. Because of feedback, execution of a microstep entails generating further events and causes firing subsequent microsteps. Events triggered during a current microstep do not influence on transitions being realized but are only allowed to affect the behavior of a controller in the next tick of discrete time: that is, in



$$E_z = \{a,b,c,d\}$$

$$X = \{a\}$$

$$Y = \{d\}$$

Figure 7-1. Statechart diagram.

the next microstep. A sequence of subsequently generated microsteps is called a *step*, and additionally it is assumed that during a step no events can come from the outside world. A step is said to be finished when there are no enabled transitions. Figure 7-3 depicts a step which consists of two simple microsteps. After the step is finished the system is in the STOP state. Summarizing, dynamic characteristics of hardware implementation are as follows:

- system is synchronous,
- system reacts to the set of available events through transition executions, and
- generated events are accessible for the system during next tick of the clock.

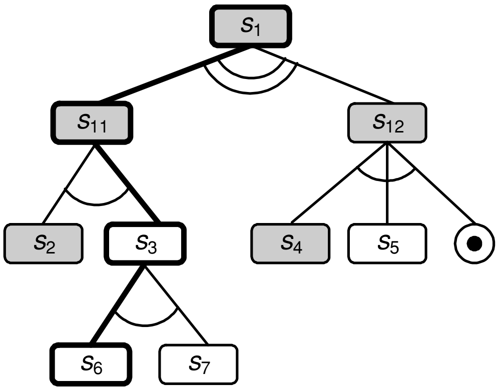


Figure 7-2. Hierarchy graph.

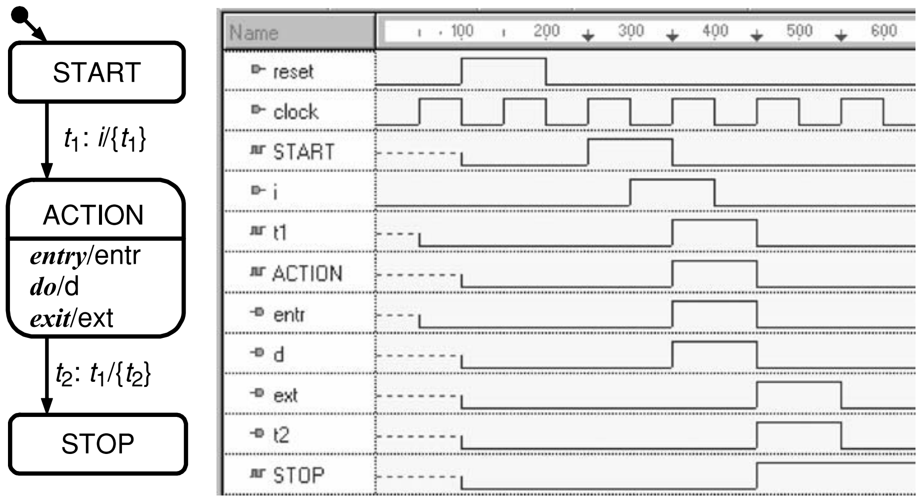


Figure 7-3. Simple diagram and its waveform.

In Fig. 7-3 a simple diagram and its waveforms illustrate the assumed dynamics features. When transition t_1 is fired ($T = 350$), event t_1 is broadcast and becomes available for the system at the next instant of discrete time ($T = 450$). The activity moves from the START to the ACTION state. Now transition t_2 becomes enabled. Its source state is active and predicates imposed on it (event t_1) are met. So, at instant of time $T = 450$ the system shifts its activity to the STOP state and triggers event t_2, which does not affect any other transition. The step is finished.

4. HARDWARE MAPPING

The main assumption of a hardware implemented behavior described with statechart diagrams is that the systems specified in this way can directly be mapped into programmable logic devices. This means that elements from a diagram (for example, states or events) are to be in direct correspondence with resources available in a programmable device~~ma~~inly flip-flops and the programmable combinatorial logic. On the basis of this assumption and taking into account the assumed dynamic characteristics, the following principles of hardware implementation have been formulated:

- Each state is assigned one flip-flop~~ac~~tivity means that the state associated with the flip-flop can be active or in the case of a state with a history attribute, its past activity is remembered; an activity of state is established on the basis of activity of flip-flops assigned to superordinate states (in the sense of a hierarchy tree).

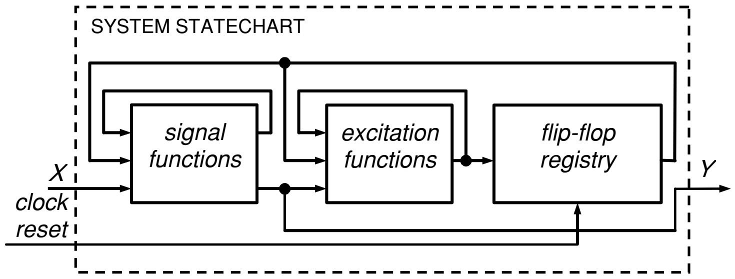


Figure 7-4. Statechart system model.

- Each event is also assigned one flip-flop~~ac~~tivity means the occurrence of an associated event and is sustained to the next tick of discrete time when the event becomes available for the system.
- On the basis of the diagram topography and rules of transition executions, excitation functions are created for each flip-flop in a circuit.

Figure 7-4 presents the main blocks of hardware implementation of statecharts.

5. GLOBAL STATE, CONFIGURATION, AND CHARACTERISITIC FUNCTION

To investigate symbolically dynamic characteristics of digital controllers specified with statechart diagrams, it is necessary to introduce notions of global state, configuration, and characteristic function.

Definition 1. Global state
Global state G is a set of all flip-flops in the system, bound both with local states and with distributed events, which can be generated in several parts of the diagram and separately memorized. ■

The diagram from Fig. 7-1 consists of 10 state flip-flops (s_1, s_{11}, s_{12}, s_2, s_3, s_4, s_5, s_6, s_7, s_e) and three event flip-flops (e_1, e_2, e_3). The flip-flop denoted as e_1 corresponds to exit event e_1 assigned to state s_2, e_2 corresponds to the entry action (e_2) to state s_5, and e_3 corresponds to the transition action (broadcasting of e_2 event) bound with transition t_5 firing (from state s_7 to state s_6). Global states comprise all information about the statechart, about both currently active states and their past activity.

An activity of a state flip-flop does not mean activity of a state bound with the flip-flop. Logic **1** on flip-flop output means actual or recent state activity. Hence, state activity is established on the basis of activity of flip-flops assigned to the superordinate states. The state is said to be active when every flip-flop bound with the states belonging to the path (in the sense of a hierarchy tree) carried from the state to the root state (located on top of a hierarchy) is asserted. Formally, a state activity condition is calculated according to the following definition:

Definition 2. State activity condition
State activity condition, denoted as *activecond(s)*, is calculated as follows:

$$activecond(s) = \prod_{s_i \in path(root_z, s)} s_i \tag{1}$$

where s_i is a signal from the flip-flop output and $path(root_z, s)$ is a set of states belonging to the path carried between $root_z$ and $s$in a hierarchy tree. ■

For example, $activecond(s_6)= s_1{}^*s_{11}{}^*s_3{}^*s_6$ (cf. Fig. 7-2, where this path is thickened).

Definition 3. Configuration
Configuration is a set of currently active states obtained in the consequence of iterative executing of the system, starting from default global state G_0. ■

A configuration carries only the information about the current state of the system and is calculated by means of the state activity condition (cf. Definition 2). For example, configuration $s_1s_{11}s_{12}s_2\bar{s}_3\bar{s}_4\bar{s}_5\bar{s}_6\bar{s}_7s_e$ corresponds to global state $s_1s_{11}s_{12}s_2\bar{s}_3\bar{s}_4\bar{s}_5s_6\bar{s}_7s_ee_1\bar{e}_2\bar{e}_3$. In Fig. 7-2 states belonging to the default configuration C_0 are grayed.

From the point of view of symbolic analysis techniques it is essential to express the concept of the set of states. The notion of the characteristic function, well known in algebra theory, can be applied.

Definition 4. Characteristic function
A *characteristic function* X_A of a set of elements $A \subseteq U$ is a Boolean function $X_A : U \rightarrow \{0, 1\}$ defined as follows:

$$X_A(x) = \begin{cases} 1 & \Leftrightarrow x \in A, \\ 0 & otherwise. \end{cases} \tag{2}$$

■

The characteristic function is calculated as a disjunction of all elements of A. Operations on sets are in direct correspondence with operations on their characteristic functions. Thus,

$$X_{(A\cup B)} = X_A + X_B; \quad X_{(A\cap B)} = X_A{}^*X_B; \quad X_{(\bar{A})} = \overline{X_A}; \quad X_{(\emptyset} = 0; \tag{3}$$

The characteristic function allows sets to be represented by binary decision diagrams[13] (BDDs). Figure 7-5 presents the characteristic function of all

$$\begin{aligned} X_{[G_0\rangle} = {} & s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 \bar{s}_6 s_7 s_e e_1 \bar{e}_2 \bar{e}_3 + s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 s_6 \bar{s}_7 s_e \bar{s}_1 \bar{e}_2 \bar{e}_3 + \\ & s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 s_6 \bar{s}_7 s_e e_1 \bar{e}_2 \bar{e}_3 + s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 s_6 \bar{s}_7 s_e e_1 \bar{e}_2 e_3 + \\ & s_1 s_{11} s_{12} s_2 \bar{s}_3 \bar{s}_4 \bar{s}_5 s_6 \bar{s}_7 s_e e_1 \bar{e}_2 \bar{e}_3 + s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 s_5 s_6 \bar{s}_7 \bar{s}_e \bar{e}_1 \bar{e}_2 \bar{e}_3 + \\ & s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 s_5 s_6 \bar{s}_7 \bar{s}_e e_1 \bar{e}_2 \bar{e}_3 + s_1 s_{11} s_{12} s_2 \bar{s}_3 \bar{s}_4 s_5 \bar{s}_6 \bar{s}_7 \bar{s}_e e_1 e_2 \bar{e}_3 + \\ & s_1 s_{11} s_{12} s_2 \bar{s}_3 s_4 \bar{s}_5 \bar{s}_6 \bar{s}_7 \bar{s}_e e_1 \bar{e}_2 \bar{e}_3 \end{aligned}$$

Figure 7-5. Characteristic function of all global states of the diagram from Fig. 7-1.

possible global states, and Fig. 7-6 the characteristic function of all configurations for the statechart from Fig. 7-1.

6. STATECHARTS SYMBOLIC STATE SPACE TRAVERSAL

Symbolic state space exploration techniques are widely used in the area of synthesis, testing, and verification of finite state systems (for example, Biliński[6]). Coudert et al.[8] were the first to realize that BDDs could be used to represent sets of states. This led to the formulation of an algorithm that traverses the state transition graph in breadth-first manner, moving from a set of current states to the set of its fan-out states. In this approach, sets of states are represented by means of characteristic functions. The key operation required for traversal is the computation of the range of a function, given a subset of its domain. The computational cost of these symbolic techniques depends on the cost of the operations performed on the BDDs and depends indirectly on the number of states and transitions. For example, the BDD characteristic function for the set of all global states for the diagram from Fig. 7-1 consists of 43 nodes, and the characteristic function for the set of all configurations has 31 nodes. The symbolic state exploration of statecharts consists in[14]

- association of excitation functions to state flip-flops,
- association of excitation functions to event flip-flops,
- association of logic functions to signals,
- representation of the Boolean function as BDDs,

$$\begin{aligned} X_{[C_0\rangle} = {} & s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 \bar{s}_6 s_7 s_e + s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 \bar{s}_5 s_6 \bar{s}_7 s_e + \\ & s_1 s_{11} s_{12} s_2 \bar{s}_3 \bar{s}_4 \bar{s}_5 \bar{s}_6 \bar{s}_7 s_e + s_1 s_{11} s_{12} \bar{s}_2 s_3 \bar{s}_4 s_5 s_6 \bar{s}_7 \bar{s}_e + \\ & s_1 s_{11} s_{12} s_2 \bar{s}_3 \bar{s}_4 s_5 \bar{s}_6 \bar{s}_7 \bar{s}_e + s_1 s_{11} s_{12} s_2 \bar{s}_3 s_4 \bar{s}_5 \bar{s}_6 \bar{s}_7 \bar{s}_e \end{aligned}$$

Figure 7-6. Characteristic function of all configurations of the diagram from Fig. 7-1.

```
symbolic_traversal_of_Statechart(Z,initial_marking) {
  $X_{[G_0\rangle}$ = current_marking = initial_marking;
  while (current_marking != $\emptyset$) {
    next_marking = image_computation(Z,current_marking);
    current_marking = next_marking * $\overline{X_{[G_0\rangle}}$ ;
      $X_{[G_0\rangle}$ = current_marking + $X_{[G_0\rangle}$ ;
  }
}
```

Figure 7-7. The symbolic traversal of statecharts.

- representation of sets of states using their characteristic functions, and
- computation of a set of flip-flop states as an image of the state transition function on the current states set for all input signals.

Starting from the default global state and the set of signals, symbolic state exploration methods enable the computation of the entire set of next global states in one formal step. Burch et al. and Coudert et al. were the first to independently propose the approach to the image computation[8,9]. Two main methods are the transition relation and transition function. The latter is the method implemented by the author. The symbolic state space algorithm of statechart ***Z*** is given in Fig. 7-7.

The variables in italics represent characteristic functions of corresponding sets of configurations. All logical variables are represented by BDDs. Several subsequent global states are simultaneously calculated using the characteristic function of current global states and transition functions. This computation is realized by the *image_computation* function. The set of subsequent global states is calculated from the following equations:

$$\begin{aligned} &\mathit{next\_marking}\\ &\quad = \exists_s \exists_x \left(\mathit{current\_marking}^* \prod\nolimits_{i=1}^{n} \left[s_i' \odot \mathit{current\_marking}^* \delta_i(s,x)\right]\right) \end{aligned} \tag{4}$$

$$\mathit{next\_marking} = \mathit{next\_marking}\langle s' \leftarrow s\rangle \tag{5}$$

where s, s', and x denote the present state, next state, and input signal respectively; $\exists_s$ and $\exists_x$ represent the existential quantification of the present state and signal variables; symbols $\odot$ and * represent logic operators XNOR and AND, respectively; equation (5) means swapping variables in the expression.

Given the characteristic function of all reachable global states of a system (see, for example, Fig. 7-5), it is possible to calculate the set of all configurations.

As mentioned earlier in Definition 2, a state is said to be active when every state belonging to the path, carried from it to the root state, is active. This led to the formulation of a state activating function. Let $activecond_i$ be a Boolean function $activecond_i : S_Z \rightarrow \{0, 1\}$, which evaluates to 1 when state s_i is active. The generation of a set of all configurations consists in the image computation of a global states characteristic function in transformation by activating function:

$$X_{[c_0\rangle} = \exists_s \exists_x \left(X_{[G_0\rangle}{}^* \prod\nolimits_{i=1}^{n} \left[s'_i \odot \left(X_{[G_0\rangle}{}^* activecond(s_i) \right) \right] \right) \tag{6}$$

$$X_{[c_0\rangle} = X_{[c_0\rangle} \langle s' \leftarrow s \rangle \tag{7}$$

In the characteristic function $X_{[c_0\rangle}$ from Fig. 7-6, s_i denotes activity of ith state. The statechart from Fig. 7-1 describes the behavior that comprises 9 global states and 6 configurations.

7. SYSTEM HICOS

Up till now, there have not been many CAD programs that employ statechart diagrams in digital circuit design implemented in programmable devices. The most prominent is *STATEMATE Magnum* by *I-Logix*[15], where the modeled behavior is ultimately described in HDL language (VHDL or Verilog) with the use of *case* and sequential instructions like *process* or *always*.

System HiCoS[16] (Fig. 7-8) automatically converts a statechart description of behavior into a register transfer level description. The input model described is in its own textual representation (statecharts specification format), which is equivalent to a graphical form, and the next is transformed into Boolean equations. Boolean equations are internally represented by means of BDDs[13,17]. Next, a reachability graph can be built or through RTL-VHDL designed model can be implemented in programmable logic devices.

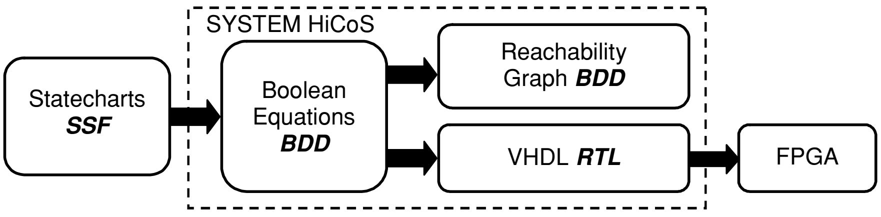


Figure 7-8. System HiCos~~sc~~hematic diagram.

8. CONCLUSION

A visual formalism proposed by David Harel can be effectively used to specify the behavior of digital controllers. Controllers specified in this way can subsequently be synthesized in FPGA circuits. In this paper it has been shown that state space traversal techniques from the FSM and Petri nets theory can be efficiently used in the fields of statechart controllers design. Within the framework of the research, a software system called HiCoS has been developed, where the algorithms have been successfully implemented.

ACKNOWLEDGMENT

The research was supported by the Polish State Committee for Scientific Research (Komitet Badań Naukowych) grant 4T11C 006 24.

REFERENCES

1. D. Harel, Statecharts, A visual formalism for complex systems. *Science of Computer Programming*, Vol. 8. North-Holland, Amsterdam, pp. 231–274 (1987).
2. G. Łabiak, Implementacja sieci Statechart w reprogramowalnej strukturze FPGA. *Mat. I Krajowej Konf. Nauk. Reprogramowalne Układy Cyfrowe*, Szczecin, pp. 169–177 (1998).
3. A. Magiollo-Schettini, M. Merro, *Priorities in Statecharts*, Diparamiento di Informatica, Universita di Pisa, Corso Italia.
4. M. Rausch B.H. Krogh, Symbolic verification of stateflow logic. In: *Proceedings of the 4th Workshop on Discrete Event System*, Cagliari, Italy, pp. 489–494 (1998).
5. UML 1.3 Documentation, Rational Software Corp. 99, http:// www.rational.com/uml
6. K. Biliński, Application of Petri Nets in parallel controllers design. PhD. Thesis, University of Bristol, Bristol (1996).
7. J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, Sequential circuit verification using symbolic model checking. In: *Proceedings of the 27th Design Automation Conference*, pp. 46–51 (June 1990).
8. O. Coudert, C. Berthet, J.C. Madre, Verification of sequential machines using Boolean functional vectors. In: *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 111–128 (November 1989).
9. A. Ghosh, S. Devadas, A.R. Newton, *Sequential Logic Testing and Verification*. Kluwer Academic Publisher, Boston (1992).
10. M. Adamski , SFC, Petri nets and application specific logic controllers. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Diego, USA, pp. 728–733 (November 1998).
11. M. von der Beeck, *A Comparison of Statecharts Variants*, LNCS, Vol. 860. Springer, pp. 128–148 (1994).
12. G. Łabiak, Wykorzystanie hierarchicznego modelu współbieżnego automatu w projektowaniu sterowników cyfrowych. PhD Thesis, Warsaw University of Technology, Warsaw (June, 2003).

13. S.-I. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publisher, Boston (1996).
14. G. Łabiak, Symbolic states exploration of controllers specified by means of statecharts. In. *Proceedings of the International Workshop DESDes'01*, Przytok pp. 209–214 (2001).
15. http://www.ilogix.com/products/magnum/index.cfm
16. http://www.uz.zgora.pl/∼glabiak
17. F. Somenzi, CUDD: CU decision diagram package, http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html

Chapter 8

CALCULATING STATE SPACES OF HIERARCHICAL PETRI NETS USING BDD

Piotr Miczulski
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgórna 50, 65-246 Zielona Góra, Poland; e-mail: P.Miczulski@iie.zu.zgora.pl

Abstract: The state space of a hierarchical Petri net can be presented as a hierarchical reachability graph. However, the hierarchical reachability graph can be described with the help of logic functions. On the other hand, binary decision diagrams (BDD) are efficient data structures for representing logic functions. Because of the exponential growth of the number of states in Petri nets, it is difficult to process the whole state space. Therefore the abstraction method of selected macromodules gives the possibility of analysis and synthesis for more complex systems. The goal of the paper is to show a method for representing the state space in the form of a connected system of binary decision diagrams as well as its calculation algorithm.

Key words: hierarchical Petri nets; state space calculation algorithm; binary decision diagram; connected system of binary decision diagrams.

1. INTRODUCTION

Hierarchical interpreted Petri nets are a hierarchical method of describing concurrent processes. They enable the design of a complex system through abstracting some parts of the net. It is possible when the abstraction part of the net is formally correct; i.e., it is safe, live, and persistent[1]. This approach can also be used for a level of the state space of a digital circuit. One of the possibilities of state spaces representation is a hierarchical reachability graph. It describes the state space on various hierarchy levels. The hierarchical reachability graph can be represented in the form of logic functions[2], in which the logic variables correspond to places and macroplaces of a Petri net. The number of them equals the number of places and macroplaces. However, the efficient methods of representing logic functions are decision diagrams, e.g., binary decision diagrams (BDDs), zero-suppressed binary decision diagrams (ZBDDs), or Kronecker functional decision diagrams (KFDDs). Therefore, each level of the hierarchy can be represented as a logic function (decision diagram), and the set of these functions creates the system of the connected decision diagrams. They describe the hierarchical state space.

In this paper, the calculation method of hierarchical state space with the help of operations on the logic functions and decision diagrams is presented. The symbolic traversal method of the state space, for flat Petri nets, was presented by Biliński[3]. The application of this method for the hierarchical Petri nets and the method of describing a hierarchical reachability graph in the form of the connected system of decision diagrams are a new idea.

2. HIERARCHICAL PETRI NETS

A hierarchical Petri net is a directed graph, which has three types of nodes called places (represented by circles), transitions (represented by bars or boxes), and macroplaces (represented by double circles). The macroplaces include other places, transitions, and also macroplaces and signify lower levels of hierarchy. When a hierarchical Petri net is used to model a parallel controller, each place and macroplace represents a local state of the digital circuit. Every marked place or macroplace represents an active local state. The set of places, which are marked in the same time, represents the global state of the controller. However, the transitions describe the events, which occur in the controller. The controller can also receive input signals coming from a data path, as well as from another control unit. It generates, using this information, control signals, which determine the behavior of the system. Input signals can be assigned to transitions in the form of logic functions. These functions are called transition predicates. If the predicate is satisfied and all input places of the transition have markers, the transition will fire.

Figure 8-1 presents an example of a hierarchical Petri net, which consists of some levels of hierarchy. The top hierarchy level is composed of macroplaces M_1 and M_5. The state space of it can be described in the form of the logic function $\chi(M_1, M_5) = M_1\overline{M}_5 + \overline{M}_1 M_5$. The lower level of the hierarchy is composed of three parallel macroplaces M_2, M_3, and M_4, which are a part of the macroplace M_5. However, the macroplaces M_1, M_2, M_3, and M_4 form the lowest level of the hierarchy. As in the top hierarchy level, every remaining abstraction level can be described with the help of a logic function.

The first step, which the designer has to do, is to create a graphical or a textual description of a Petri net. With this end in view, the designer can use

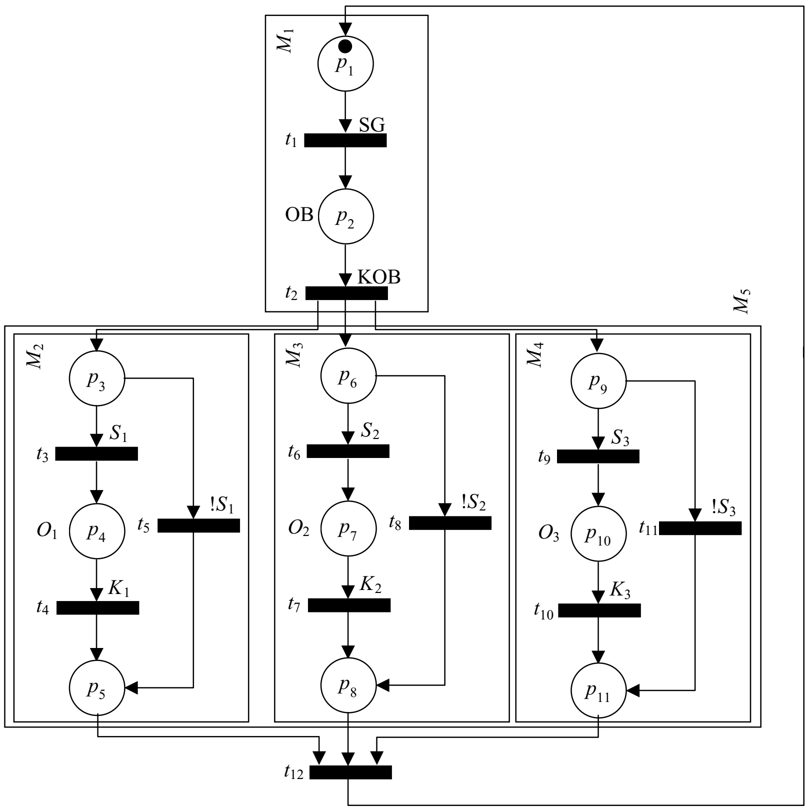


Figure 8-1. An example of a hierarchical Petri net.

the textual format PNSF2 for a specification of the flat and hierarchical Petri nets[4]. In the next step, the computer program loads this specification to internal object data structures. During loading the textual or graphical description of a Petri net (or nets), the basic validation of the Petri net structure is made. The class diagram of the designed Petri net object library (PNOL) is described in the UML notation, in Fig. 8-2.

After reading a structure of the Petri net, the space of the states of a digital controller can be calculated. It can be done on the grounds of an internal data structure of the Petri net and with the help of BDDs.

3. THE STATE SPACE AND BINARY DECISION DIAGRAM

A binary decision diagram (BDD) is a rooted, directed, acyclic graph, which has two sink nodes labeled 0 and 1, representing the Boolean function values

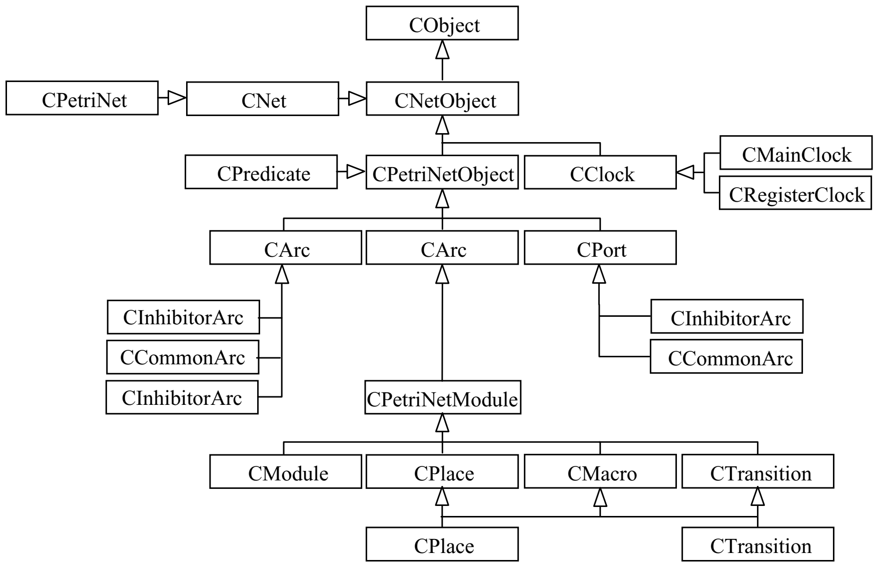


Figure 8-2. The class diagram of the Petri net object library.

0 and 1, and non-sink nodes, each labeled with a Boolean variable. Each non-sink node has two output edges labeled 0 and 1 and represents the Boolean function corresponding to its edge 0 or the Boolean function corresponding to its edge 1. The construction of a standard BDD is based on the Shannon expansion of a Boolean function[5,6]. An ordered binary decision diagram (OBDD) is a BDD diagram in which all the variables are ordered and every path from the root node to a sink node visits the variables in the same order. A reduced ordered binary decision diagram (ROBDD) is an OBDD diagram in which each node represents a distinct logic function. The size of a ROBDD greatly depends on the variable ordering. Many heuristics have been developed to optimize the size of BDDs[5,6]. In this paper, all binary decision diagrams are reduced and ordered.

The whole state space of the hierarchical Petri net (Fig. 8-1) can be described as one logic function:

$$\begin{aligned}
&\chi(p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}) = \\
&p_1\overline{p_2p_3p_4p_5p_6p_7p_8p_9p_{10}p_{11}} + \overline{p_1}p_2\overline{p_3p_4p_5p_6p_7p_8p_9p_{10}p_{11}} + \\
&\overline{p_1p_2}p_3\overline{p_4p_5}p_6\overline{p_7p_8}p_9\overline{p_{10}p_{11}} + \overline{p_1p_2p_3}p_4\overline{p_5}p_6\overline{p_7p_8}p_9\overline{p_{10}p_{11}} + \\
&\overline{p_1p_2}p_3\overline{p_4p_5p_6}p_7\overline{p_8}p_9\overline{p_{10}p_{11}} + \overline{p_1p_2}p_3\overline{p_4p_5}p_6\overline{p_7p_8p_9}p_{10}\overline{p_{11}} + \\
&\overline{p_1p_2p_3p_4}p_5p_6\overline{p_7p_8}p_9\overline{p_{10}p_{11}} + \overline{p_1p_2}p_3\overline{p_4p_5p_6p_7}p_8p_9\overline{p_{10}p_{11}} + \\
&\overline{p_1p_2}p_3\overline{p_4p_5}p_6\overline{p_7p_8p_9p_{10}}p_{11} + \overline{p_1p_2p_3}p_4\overline{p_5}p_6\overline{p_7p_8p_9}p_{10}\overline{p_{11}} + \\
&\overline{p_1p_2p_3}p_4\overline{p_5p_6}p_7\overline{p_8}p_9\overline{p_{10}p_{11}} + \overline{p_1p_2p_3}p_4\overline{p_5p_6p_7}p_8p_9\overline{p_{10}p_{11}} +
\end{aligned}$$

$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,p_4\,\overline{p_5}\,p_6\,\overline{p_7}\,\overline{p_8}\,\overline{p_9}\,\overline{p_{10}}\,p_{11} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,p_4\,\overline{p_5}\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,p_{10}\,\overline{p_{11}} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,p_4\,\overline{p_5}\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,p_{10}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,\overline{p_7}\,p_8\,p_9\,\overline{p_{10}}\,\overline{p_{11}} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,p_7\,\overline{p_8}\,p_9\,\overline{p_{10}}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,\overline{p_{10}}\,p_{11} +$$
$$\overline{p_1}\,\overline{p_2}\,p_3\,\overline{p_4}\,\overline{p_5}\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,p_{10}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,p_3\,\overline{p_4}\,\overline{p_5}\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,\overline{p_{10}}\,p_{11} +$$
$$\overline{p_1}\,\overline{p_2}\,p_3\,\overline{p_4}\,\overline{p_5}\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,p_{10}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,p_6\,\overline{p_7}\,\overline{p_8}\,\overline{p_9}\,\overline{p_{10}}\,p_{11} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,p_6\,\overline{p_7}\,\overline{p_8}\,\overline{p_9}\,p_{10}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,p_3\,\overline{p_4}\,\overline{p_5}\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,\overline{p_{10}}\,p_{11} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,p_4\,\overline{p_5}\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,\overline{p_{10}}\,p_{11} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,p_4\,\overline{p_5}\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,\overline{p_{10}}\,p_{11} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,p_{10}\,\overline{p_{11}} + \overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,\overline{p_7}\,p_8\,\overline{p_9}\,p_{10}\,\overline{p_{11}} +$$
$$\overline{p_1}\,\overline{p_2}\,\overline{p_3}\,\overline{p_4}\,p_5\,\overline{p_6}\,p_7\,\overline{p_8}\,\overline{p_9}\,\overline{p_{10}}\,p_{11}$$

This means that the modeling controller may be in one of 29 states. The BBD diagram, for this function, has 24 non-sink nodes. We can reduce the number of nodes by creating a connected system of BDDs (see Fig. 8-3). Each decision diagram describes one characteristic function of the appropriate hierarchy level. In this case there are six decision diagrams, which have 19 non-sink nodes. This situation follows from the fact that the size of decision diagram depends, among other things, on the number of logic variables of the function. Besides, each

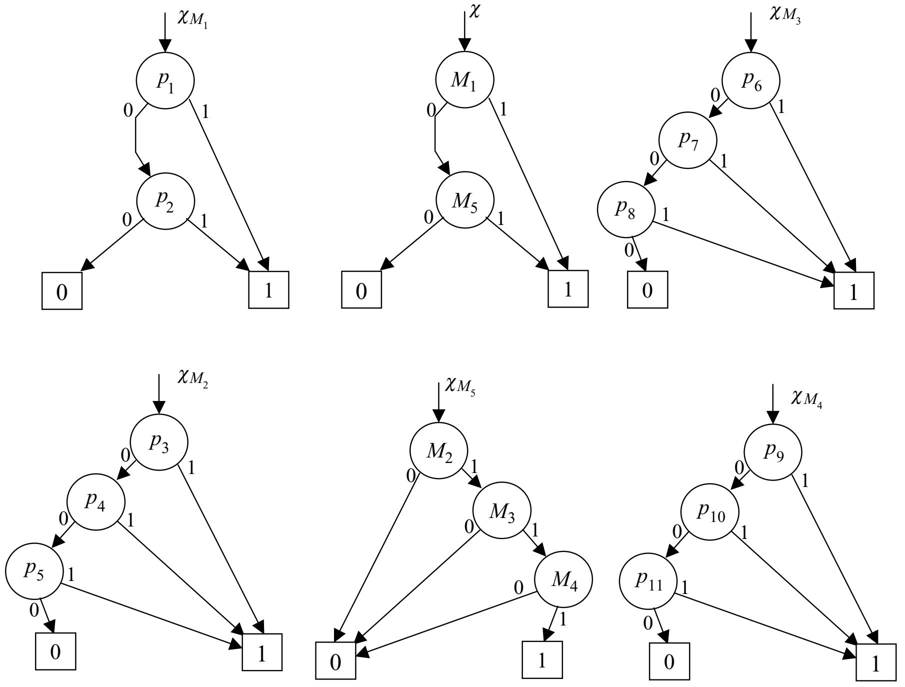


Figure 8-3. The connected system of the binary decision diagrams.

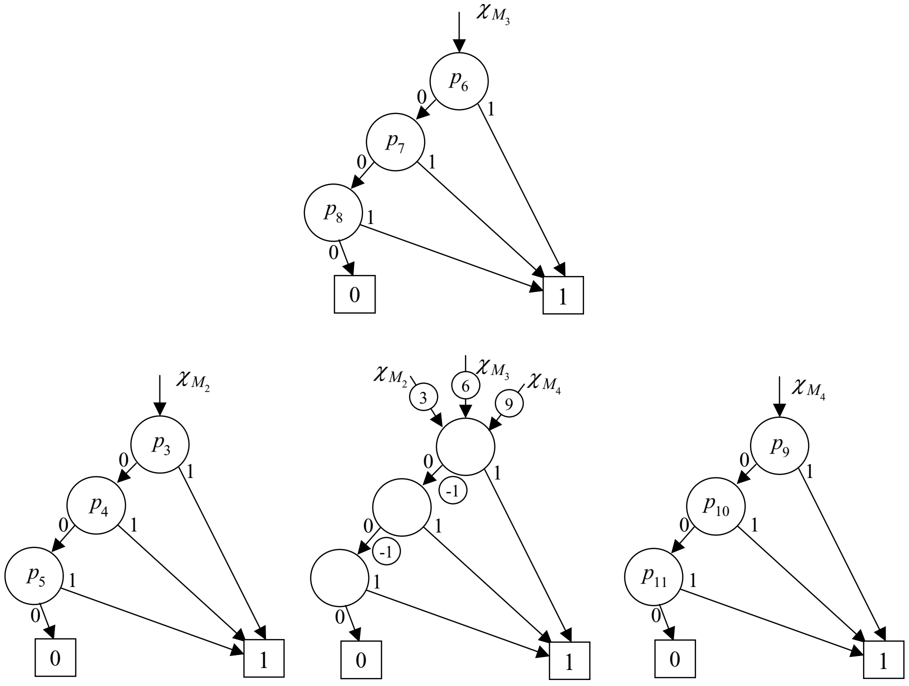


Figure 8-4. Variable shifters in the connected system of the decision diagrams.

logic function can be represented with the help of another kind of the decision diagram. The kind of the decision diagram used strictly depends on the kind of the function. For example, for one logical function the binary decision diagram is better, but the zero-suppressed binary decision diagram is better for another function.

In many cases we can find similar parts of the Petri net, for which there is only one difference: a naming of the places. It means that they are isomorphic. Therefore, the state spaces of similar parts of the Petri net can be represented by the same BDD, with the special edge attributes, called variable shifters (see Fig. 8-4). These edge attributes were described by Minato[6]. They allow a composition of some decision diagrams into one graph by storing the difference of the indices. The variable shifter edges indicate that a number should be added to the indices of all descendant nodes. The variable index is not stored with each node, because the variable shifter gives information about the distance between the pair of nodes. In the case of the edges, which point to a terminal node, a variable shifter is not used. The variable shifters, which point to the root of a BDD, indicate the absolute index of the node. The other variable shifters, which are assigned to the edges of the BDD, indicate the difference between the index of the start node and the end node. We can use these edges to reduce the number of nodes in the connected system of the decision diagrams, and it will allow designing a more complex system.

4. ALGORITHM OF STATE SPACE CALCULATION

After parsing of a Petri net description, written in the PNSF2 format, and loading a Petri net to internal data structures, the algorithm checks the structure of the Petri net (see Fig. 8-5). Afterward, if it is a flat Petri net, the algorithm

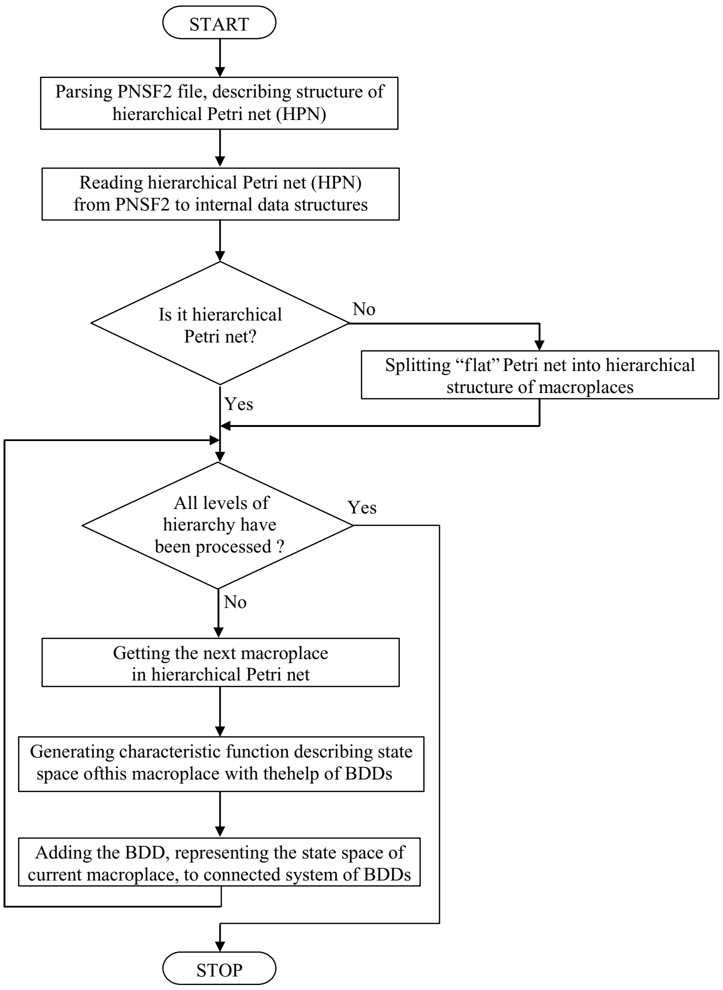

START
Parsing PNSF2 file, describing structure of hierarchical Petri net (HPN)
Reading hierarchical Petri net (HPN) from PNSF2 to internal data structures
Is it hierarchical Petri net?
No
Splitting "flat" Petri net into hierarchical structure of macroplaces
Yes
All levels of hierarchy have been processed ?
Yes
No
Getting the next macroplace in hierarchical Petri net
Generating characteristic function describing state space ofthis macroplace with thehelp of BDDs
Adding the BDD, representing the state space of current macroplace, to connected system of BDDs
STOP

Figure 8-5. Calculation algorithm of the state space of a hierarchical Petri net.

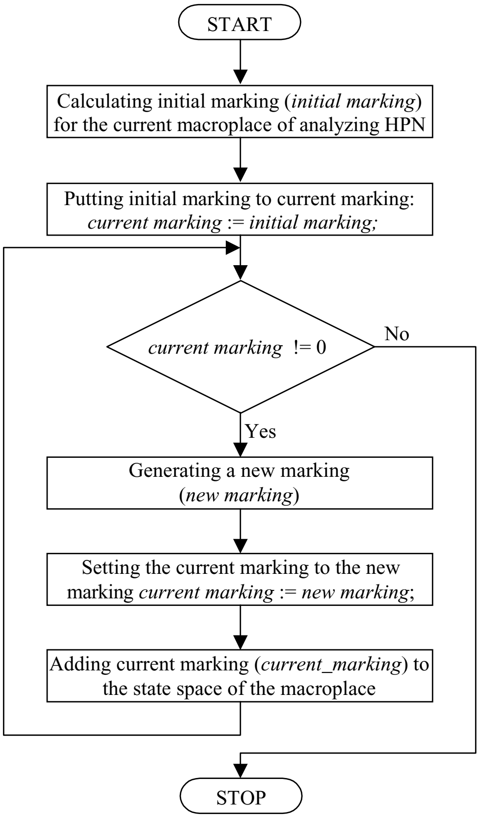

START
Calculating initial marking (*initial marking*) for the current macroplace of analyzing HPN
Putting initial marking to current marking: *current marking* := *initial marking;*
current marking != 0
No
Yes
Generating a new marking (*new marking*)
Setting the current marking to the new marking *current marking* := *new marking*;
Adding current marking (*current_marking*) to the state space of the macroplace
STOP

Figure 8-6. Symbolic traversal algorithm for a hierarchical Petri net.

splits it into a hierarchical structure of macroplaces. In the next step, for each macroplace, the algorithm recursively calculates characteristic function. Each logic function, represented in the form of a decision diagram, describes a state space of each macroplace. However the calculated decision diagram is joined to the connected system of decision diagrams, which represents the whole state space of the hierarchical Petri net. During this process we can also check whether the Petri net is formally correct.

One of the more important steps of this algorithm is the calculation of the characteristic function, which describes the state space of a macroplace (Fig. 8-6). The symbolic traversal algorithm is taken from Biliński[3]. In this method, the next marking is calculated using the characteristic function and the transition function. Transition functions ($\Delta : \Omega \rightarrow \Omega$) are the logic functions associated with places and defined as a functional vector of Boolean functions: $\Delta = [\delta_1(P, X), \delta_2(P, X), \ldots, \delta_n(P, X)]$, where $\delta_i(P, X)$ is a transition function of place p_i; P and X are sets of places and input signals, respectively.

The function δ_i has value 1 when place p_i has a token in the next iteration; otherwise it equals 0. Every function δ_i consists of two parts: a part describing the situation when the place p_i will receive a token and a part describing the situation when the place will keep a token. For example, place p_7 will have a token in the next iteration if place p_6 has a token and input signal S_2 is active (transition t_6 will fire) or place p_7 has already got a token and input signal K_2 is inactive (transition t_7 is disabled); thus the function δ_7 can be defined as follows: $\delta_7 = p_6 * S_2 + p_7 * \overline{K}_2$.

The computation of a set of markings which can be reached from the current marking (*current_marking*) in one iteration is carried out according to the following equations:

$$\mathit{next\_marking} = \underset{p}{\exists}\,\underset{x}{\exists}(\mathit{current\_marking} * \prod_{i=1}^{n}[p'_i \odot (\mathit{current\_marking} * \delta_i(p, x))])$$

where p, p', and x denote the present state, the next state, and the input signal, respectively, and symbols $\odot$ and * represent logic operators XNOR and AND, respectively.

5. SUBMISSION

The application of a connected system of binary decision diagrams enables to reduce the number of nodes of decision diagrams. On the other hand, application of the hierarchical Petri nets makes designing parallel digital controllers easier. It means that we can process more complex digital circuits on various abstraction levels without processing the whole state space. The connected system of decision diagrams can also be used for a formal verification of the hierarchical Petri nets.

ACKNOWLEDGMENT

The research was supported by the Polish State Committee for Scientific Research (Komitet Badań Naukowych) grant 4T11C 006 24.

REFERENCES

1. T. Murata, Petri nets: properties, analysis and applications. In: *Proceedings of the IEEE*, **77** (4), 541–580 (1989).

2. E. Pastor, O. Roig, J. Cortadella, M. R. Badia, Application and theory of Petri nets. In: *Proceedings of 15th International Conference, Lecture Notes in Computer Science*, Vol. 815, *Petri Net Analysis Using Boolean Manipulation*. Springer-Verlang (1994).
3. K. Biliński, Application of Petri nets in parallel controller design. PhD. Thesis, University of Bristol (1996).
4. M. Węgrzyn, Hierarchical implementation of concurrent logic controllers by means of FPGAs. PhD. Thesis, Warsaw University of Technology (1998).
5. R. Drechsler, *Binary Decision Diagram. Theory and Implementation*. Kluwer Academic Publishers, pp. 9–30 (1998).
6. S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, pp. 9–22 (1996).

Chapter 9

A NEW APPROACH TO SIMULATION OF CONCURRENT CONTROLLERS

Agnieszka Węgrzyn and Marek Węgrzyn
University of Zielona Góra, Institute of Computer Engineering and Electronics, ul. Podgorna 50, 65-246 Zielona Góra; e-mail: A.Wegrzyn@iie.uz.zgora.pl, M.Wegrzyn@iie.uz.zgora.pl

Abstract: In the paper a new approach to simulation of modeled concurrent controllers is presented. In the case presented, a concurrent controller is modeled using Petri nets and HDL languages. A very important stage of digital circuits design is verification, because it saves time and money. Simulation is the simplest method of verification. In the literature a lot of approaches to circuit simulation are described, but a new technology gives new possibilities and it gives a new idea to use XML and Verilog to simulate concurrent controllers.

Key words: Petri net; simulation; XML; HDL; verification.

1. INTRODUCTION

Dynamic expansion of electronics in daily life and constantly developing digital technology are making life easier, better, and safer. Increasing demands for digital circuits and controllers create a new working field for engineers. The designers encounter new problems and have to search new, optimal methods of design of programmable logic controllers (PLCs).

Nowadays in the world, for design and modeling digital circuits and concurrent controllers, two kinds of the models, HDL languages and Petri nets, are used most frequently. Testing of concurrent controller model is very important, because a good system should not contain events that can never occur.

There exist many different methods of systems verification, but one of the simplest is simulation. This method enables checking of behavior correctness of the model, and in the same way of a real circuit. Furthermore, the simulation allows to detect and remove many errors at an early stage of design. The

verification of modeling concurrent controllers using Petri nets can be carried out in the following three ways:
- animation of Petri nets,
- analysis of Petri nets,
- simulation of HDL model.

In the paper, two methods of simulation will be presented: animation of Petri net using Scalable Vector Graphics and simulation of Petri nets using the HDL language.

Verilog has become the standard interface between human and design automation tools. It can describe both functional and behavioral information about a digital system, as well as the structure of the interconnected composite blocks down to the physical implementation level. It can be also used for modeling and design of industrial logic controllers, especially those that are realized with modern FPGAs as application specific logic controllers (ASLCs)[6].

Verilog is used for Petri net model prototyping of industrial logic controllers. It is used for operation unit modeling of the system, too.

2. BACKGROUND

In this section basic information about and definitions of Petri nets and HDL language are presented.

2.1 Petri nets

Petri nets[1] are mathematical objects that exist independently on any physical representation and implementation. This model allows to describe designed digital circuits, especially concurrent controllers in a graphical and mathematical way. The graphical form of Petri nets is often used as a tool for analysis and modeling of digital circuits on high-level abstraction.

In the original version (which was defined in 1961 by C.A. Petri) this model enabled to specify the rules of digital circuit action, which was meant for asynchronous and nondeterministic nets. Well-known and perfect mathematical formalism enables modeling concurrent processes, such as sequential, conflict, and concurrent. Later, this model was extended to encompass other more useful functions, such as synchronization, interpretation, hierarchy, and color.

For the modeling of digital systems, selected classes of Petri nets are applicable. In this paper synchronous, interpreted, hierarchical, and colored Petri nets are described.

A Petri net model can be presented as an oriented bipartite graph with two subsets of nodes called places and transitions, where each node is linked with another by an arc. A Petri net model can be described as

$$\mathrm{PN} = (P, T, F),$$

where

- P is a finite set of places;
- T is a finite state of transitions, and
- F is a finite set of arcs.

In a graphical form of Petri nets each place represents a local state of a digital circuit. A currently active place is marked. Every marked place represents an active local state. The set of places, which are marked at the same time, defines the global state of the controller. Each transition describes logical condition of controller's state change.

In the flow graph, each place has a unique identifier and an ordered list of output signals, which are activated in this state (Moore outputs). However, each transition, besides the unique name's tag, has an enabling function to change the currently active state. This function consists of two parts: logical condition of state change and firing results. The logical condition consists of inputs product, outputs, predicates, and places, which are connected by inhibitor and enabling arcs to this transition. When a logical condition is satisfied and all input places of the transition have markers, then the transition is being fired with the nearest rising edge of a clock signal. As a result of execution, input places shift active markers to output places of the transition and appropriate output signals are activated (Mealy outputs).

In large projects, it is more useful to create models of Petri nets with the use of hierarchical approach. In this way any part of the net can be replaced by the symbol of a macronode. Each macroplace or macrotransition includes other places, transitions, arcs, and other macronodes. They are making subnets and represent lower levels of the hierarchy. When the macronode becomes active (gets a marker), the control is passed to the subnet and starting places of the subnet are marked inside. The marker stays at this macronode until all output places of the subnet are marked. Then the control is passed back to the main net.

2.2 HDL language

Even today, the usual validation method to discover design errors is simulation. The implementation is expressed in terms of a hardware description language such as Verilog HDL or VHDL, which can be simulated. By simulation it is possible to decide whether the resulting output sequence violates the specification, i.e., the implementation behaves erroneously[2].

Verilog[4] is a hardware description language used to design and document electronic systems. It allows designers to design at various levels of abstraction. It was invented as a simulation language. Use of Verilog for synthesis was a complete afterthought. The IEEE standardization process includes enhancements and refinements; at the end the work on the Verilog 1364-2001 standard was analyzed. It is a language capable of representing the hardware with the independence of its final implementation.

Verilog is increasingly used to describe digital hardware designs that are applicable for both simulation and synthesis. In addition, these HDLs are commonly used as an exchange medium, e.g., between the tools of different vendors. HDL modeling of Petri nets is an actual research topic presented in papers.

3. MODELING OF LOGIC CONTROLLERS

In this section two methods of modeling of logic controllers (especially concurrent controllers), i.e., Petri nets and Verilog-HDL language, are presented.

3.1 An example of a concurrent controller

In this section, an example of a digital controller, which is used to control in technology process of liquids mixing and transporting them is presented. A general diagram of this process is presented in Fig. 9-1. The whole process consists of three main steps.

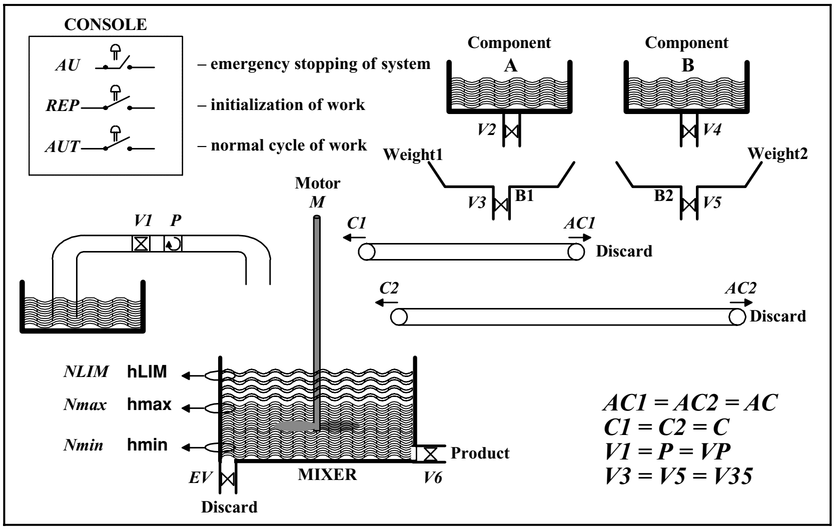


Figure 9-1. Schema of a technological process.

Table 9-1. Input and output signals

Name	Description
AU	Breakdown appearance notification
REP	Initial transaction signal
AUT	Enabling signal for starting operation
B1, B2	Notification product A and B proper weight
NLIM	Maximal level of foam (hLIM)
Nmax	Maximal level of liquid (hmax)
Nmin	Minimal level of liquid (hmin)
V1, P	Valve-opening and pump-running signals
V2, V4	Signals opening valves of containers with products A, B
V3, V5	Signals opening valves in scales for products A, B
C1, C2	Signals running conveyor belt delivering products A, B
AC1, AC2	Signals running conveyor belts removing redundant products A, B
EV	Signal opening main container valve for removing waste
V6	Signal opening main container valve for the final product

Outputs P, V1, V2, V4, AC1, AC2, EV, and M are Mealy outputs; however, outputs C1, C2, V3, V5, and V6 are Moore outputs.

The schema of the technological process is presented in Fig. 9-1. An example of the process is taken from Ref. 6.

Table 9-1 gives a description of each input and output signal. Generally, the working of a system can be divided into three stages:

- stage I – initialization of work;
- stage II – normal cycle of work;
- stage III – emergency stopping of the system.

Figure 9-2 presents Petri nets for the logic circuit that controls the described technological process. The controller is described by two nets. Each coherent net is analyzed separately. In literature there are some ways to analyze such nets, using enabling arcs. But, for the sake of the state local code P14 occurring in the condition for a lot of transitions (e.g., t2, t3, t4), insert an additional signal M14, because joining a lot of enabling arcs going out from place P14 decreases the legibility of the net.

3.2 Petri net specification format 3

New technologies give new possibilities. Since XML[7] was developed by an XML Working Group in 1996, several applications of XML to a specific domain have been created, e.g., CML for chemistry and GedML for genealogy. XML markup describes a document's structure and meaning.

The development of a new XML-based format for Petri nets enables transforming modeled Petri nets between systems that use an XML-based format.

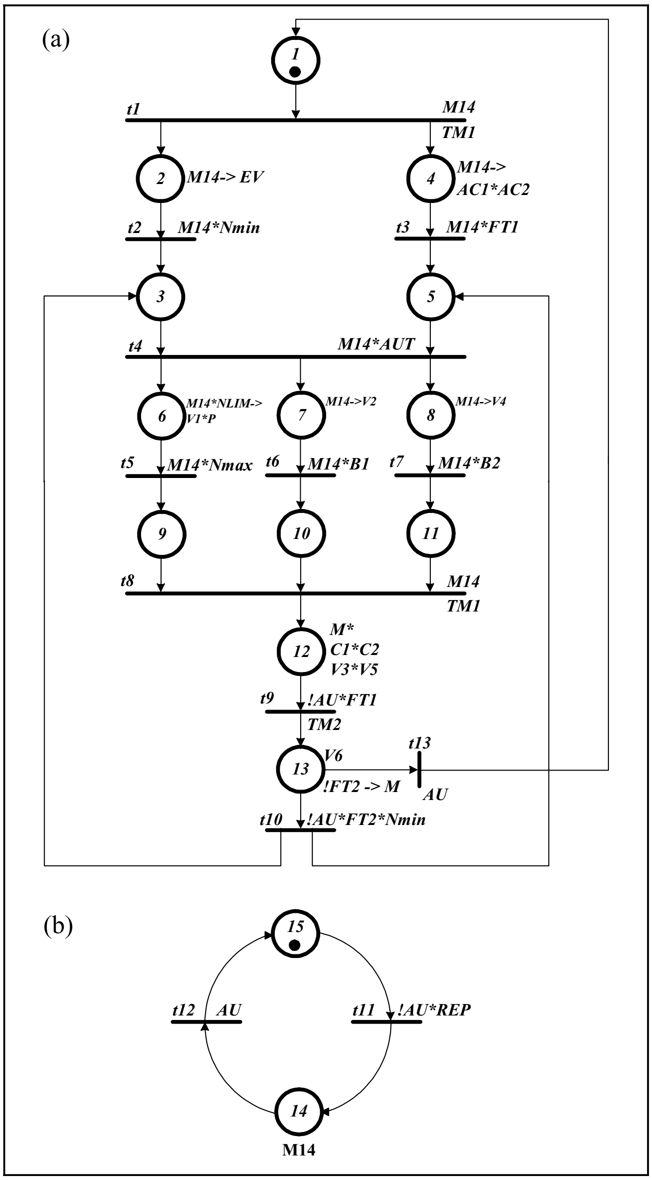


Figure 9-2. PNSF3 format for Petri nets.

Because of this fact a new XML format, PNSF3 (Petri net specification format 3), was proposed[5]. This format is based on PNSF2, which was developed at the University of Zielona Góra.

PNSF3 is one of such textual descriptions of concurrent controller models. In this format XML markup notation is used for storing information about the controller. The main goal of PNSF3 is specification of the principle of concurrent controller action. By means of PNSF3 format, interpreted, synchronous, hierarchical, and colored Petri nets can be modeled. As a result, this format is simple to create and modify.

PNSF3 specifies the structure of a Petri net. It allows describing each place and transition of the net and connection between them inside a graph. It stores information about clock, input and output signals, and their kinds. However, PNSF3 does not keep any information about the way of element presentation and placement inside the graph, as opposed to the format described in Ref 3.

Each such XML application should have its own syntax and vocabulary defined by an outside DTD file. The XML language is a set of rules defining semantic tags that break a document into parts and identify the logical parts of the document. XML markups describe only the syntax, meaning, and connections between data. This format is good for storing large and complex structures of data textual format. It is easy to read, write, and convert into other formats.

PNSF3 has its own dictionary of XML markups. The dictionary consists of a few main markups, which are grouped and provide information about the model. Each defined markup includes a list of arguments or child markups, which enable precise description of a given element of the graph. One such markup, which describes place, is presented in Fig. 9-3.

A general tag of the document, which is validated by the DTD file, is presented in Fig. 9-4.

The main advantage of XML is that it is very easy to transform into another format (e.g. SVG). In this paper, a way of transformation from the XML-based format (PNSF3) to SVG[8] is presented. The SVG file holds information about places, transitions, predicates, markings, and placement of these elements. Using the option of dynamic SVG animation, it is possible to simulate an interpreted Petri net.

A PNSF3 format (Fig. 9-5) was prepared for the Petri net (Fig. 9-2).

3.3 HDL modeling and simulation

Several methods were proposed to transfer Petri nets specifications into VHDL for performance and reliability analysis. But in the literature,

```
<PLACE ID="p1" MARKING="yes"
               ID_COLOURS="c1 c2 c3"   ID_MACROPLACE=""...>
      place_name
</PLACE>
```

Figure 9-3. An example of a PNSF3 markup.

```
<? xml version="1.0" encoding="ISO-8859-2"
                              standalone="no" ?>
<!DOCTYPE PNSF3 SYSTEM "pnsf3.dtd">
<PNSF3>
  <GLOBAL>
       declarations of global signal ....
  </GLOBAL>

  <MACRO_PLACE>
     description of macroplace used in other  units ....
  </MACRO_PLACE >
  <MACRO_TRANSITION>
     description of macrotransition used in other units
  </MACRO_TRANSITION>

  <PART>
       description of independent unit ....
  </PART>
</PNSF3>
```

Figure 9-4. A general template of a PNSF3 document.

Verilogbased modeling of Petri nets is not known. Verilog syntax can support the intermediate-level models. It makes it possible to describe the highest level of the system and its interfaces first and then to refer to greater details.

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules, as well as a description of its contents[4].

The basic Verilog statement for describing a process is the *always* statement. The *always* continuously repeats its statement, never exiting or stopping. A behavioral model may contain one or more *always* statements.

The *initial* statement is similar to the *always* statement, except that it is executed only once. The *initial* provides a means of initiating input waveforms and other simulation variables before the actual description begins its simulation. The *initial* and *always* statements are the basic constructs for describing concurrency[4].

Because of the fact that Verilog can model concurrency, e.g., using structure *always*, Petri nets can be effectively described by the Verilog language.

The conditions of transitions are input signals, or internal signals (for subnets synchronization). Each place is represented by a flip-flop (i.e., concurrent one-hot method is used for place encoding), which holds the local state of the controller. During initialization of the simulation, and after active state of signal *reset*, initial marked places are set to logical 1, other places are set to 0.

In Fig. 9-6 a part of a Verilog model is presented. At the beginning, a file called *defines.h* is included, and real names of objects (input/output signals, places, and transitions) are assigned to names used in the model. In the example presented, the signals Nmin, V1, and TM1 are assigned as follows:

```
<?xml version="1.0" encoding="ISO-8859-2"
standalone="no"?>
<!DOCTYPE pnsf3 SYSTEM "pnsf3.dtd">
<PNSF3>
 <GLOBAL>
 <CLOCK ID="clk1" > clk </CLOCK>
 <DEF_COLOURS>
   <COLOUR ID="col1" > red </COLOUR>
   <COLOUR ID="col2" > green </COLOUR>
       …
 </DEF_COLOURS>
 <INPUTS>
   <INPUT ID="i1" > x0 </INPUT>
   <INPUT ID="i2" > x1 </INPUT>
   <INPUT ID="i3" > x2 </INPUT>
       …
 </INPUTS>
 <OUTPUTS>
   <OUTPUT ID="o1" > y1 </OUTPUT>
   <OUTPUT ID="o2" > y2 </OUTPUT>
   <OUTPUT ID="o3" > y3 </OUTPUT>
       …
 </OUTPUTS>
 </GLOBAL>
 <PART NAME="sterownik" ID="part1">
 <PLACES>
   <PLACE ID="p1"  MARKING="yes"
    ID_COLOURS="col1 col2 col3 col4">  p1
   </PLACE>
   <PLACE ID="p2"  ID_COLOURS="col1 col2">  p2
   </PLACE>
   <PLACE ID="p3"  ID_COLOURS="col3">  p3
   </PLACE>
       …
 </PLACES>
<TRANSITIONS>
   <TRANSITION ID="t1"  ID_COLOURS="col1 col2 col3 col4">
    t1
   </TRANSITION>
   <TRANSITION ID="t2"  ID_COLOURS="col1 col2" >
    t2
   </TRANSITION>
   <TRANSITION ID="t3"  ID_COLOURS="col3" >
    t3
   </TRANSITION>
      …
 </TRANSITIONS>
 <NET>
   <ARC ID_TRANSITION="t1"  ID_IN_PLACES="p1"
        ID_IN_SIGNALS="i1"  ID_OUT_PLACES="p2 p3 p6"
        ID_OUT_SIGNALS="o1 o2 o9">
   </ARC>
   <ARC ID_TRANSITION="t2"  ID_IN_PLACES="p2"
        ID_IN_SIGNALS="i2"  ID_OUT_PLACES="p4">
   </ARC>
   <ARC ID_TRANSITION="t3"  ID_IN_PLACES="p3"
        ID_IN_SIGNALS="i4"  ID_OUT_PLACES="p5">
   </ARC>
       …
 </NET>
 <MOORE_OUTPUTS>
   <MOORE_DESC ID_IN_PLACES="p2" ID_OUT_SIGNALS="o1">
   </MOORE_DESC>
   <MOORE_DESC ID_IN_PLACES="p3" ID_OUT_SIGNALS="o2">
   </MOORE_DESC>
   <MOORE_DESC ID_IN_PLACES="p6" ID_OUT_SIGNALS="o9">
   </MOORE_DESC>
       …
 </MOORE_OUTPUTS>
 </PART>
```

Figure 9-5. PNSF3 format for Petri nets.

```
`include "defines.h"
module Petri_Net_example (reset, clk, INs, REGOUTs, OUTs);
//Declarations
input reset, clk;
input [9:0] INs;
output [13:0] OUTs;
output [1:0] REGOUTs; reg [1:0]  REGOUTs  ;
reg [14:0] States;
wire [0:12] T;
// Conditions for transitions
assign `T1 = `M14;
assign `T2 = `M14 & `Nmin;
assign `T3 = `M14 & `FT1;
...
//Combinatorial outputs ...
assign  `V1 = (`M14 & `NLIM==1)?`P6 :0;
assign  `V2 =  (`M14==1)?`P7:0;
assign  `V3 = `P12;
assign  `P =  ((`M14 & `NLIM)==1)?`P6:0 ;
...
//... and registered outputs
assign  `M14 =  `P14;
always @(posedge clk)
 begin
     if (reset)  `TM1 <= 0;
     else `TM1 <= `T1&`P1 |  `T8&`P9&`P10&`P11 ;
 end
always @(posedge clk)
 begin
     if (reset)  `TM2 <= 0;
     else `TM2 <= `T9&`P12 ;
 end
//Exciting functions for flip-flops (places)
always @(posedge clk)
 begin
     if (reset)  `P1 <= 1;
     else `P1 <=  (`P1 &~`T1 )|(`T13&`P13);
 end
always @(posedge clk)
 begin
     if (reset)  `P2 <= 0;
     else `P2 <=  (`P2 &  ~`T2 ) | (`P1 & `T1) ;
 end
...
```

Figure 9-6. A part of the Verilog model.

```
'define Nmin INs[0]
'define V1 OUTs[0]
'define TM1 REGOUTs[0]
```

In a module, the designed controller interface is declared, i.e., the number, types, and size of input/output ports. Then there are two *assign* blocks. The first block defines conditions, with respect to input signals, for firing of transitions. The second one defines logical outputs for each active local state of the controller. The next blocks are a group of *always* statements, which calculate the current state of the flip-flops. Each *always* statement controls one place of a net. During work of the controller, the exciting function of the flip-flop for the given state (place) can be calculated as follows:

$$P_x = P_{x}\_set + P_{x}\_hold$$

and

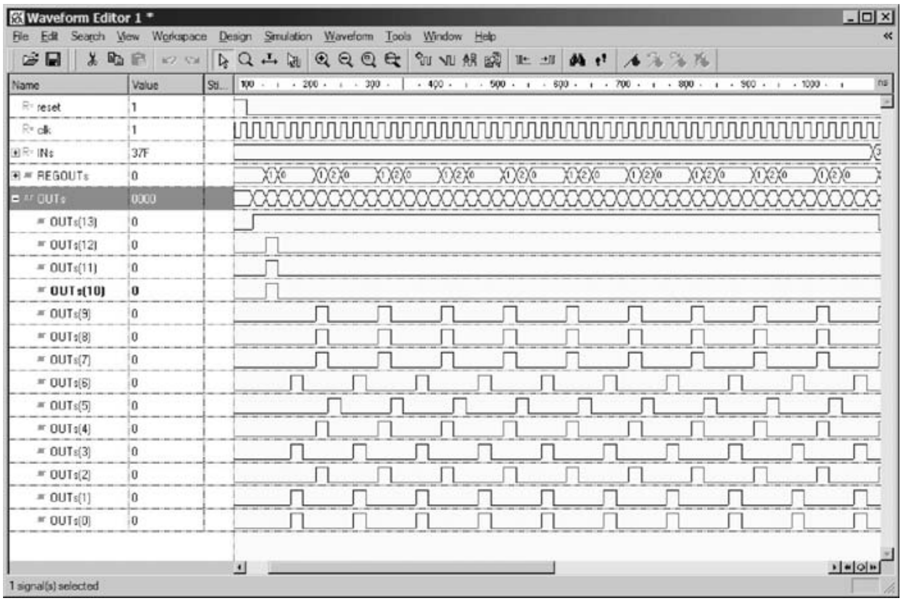
$$P_{x}\_set = \sum_{t_i \in \bullet p_x} C(t_i)$$

$$P_{x}\_hold = p_x * \prod_{t_i \in P_x \bullet} \sim C(t_i)$$

where

- $^{\bullet}P_x$ is the set of the input transitions of a place P_x;
- $P_x\bullet$ is the set of the output transitions of a place P_x;
- $C(t_i)$ is the firing condition of the transition t_i; and
- $\sum$, $\prod$, $*$, and $\sim$ are logical operations OR, AND, AND, NOT, respectively.

Figure 9-7 shows a window from Aldec Active-HDL simulator with the waveforms as the simulation result (the vector of the output signals is presented in detail).


Waveform Editor 1 *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
1 signal(s) selected

Name	Value
reset	1
clk	1
INs	37F
REGOUTs	0
OUTs	0000
OUTs[13]	0
OUTs[12]	0
OUTs[11]	0
OUTs[10]	0
OUTs[9]	0
OUTs[8]	0
OUTs[7]	0
OUTs[6]	0
OUTs[5]	0
OUTs[4]	0
OUTs[3]	0
OUTs[2]	0
OUTs[1]	0
OUTs[0]	0

Figure 9-7. Results of the Verilog model simulation.

4. SIMULATION OF THE PETRI NET MODEL USING SVG

The next step after modeling is analysis of prepared specification. Scalable Vector Graphics (SVG) is a format of vector graphics that is based on XML language. The SVG format is supported by some script languages for animation. It gives an idea for using such a format for presenting Petri nets in a graphical form.

SVG was created by World Wide Web Consortium (W3C). SVG is a language for describing two-dimensional (2D) vector graphics by means of XML markup notation. This format has its own vocabulary of restricted and defined meanings, recognizable by Web browser tools. SVG has many advantages in comparison with other raster graphic formats commonly used on the Web today, such as GIF or JPEG, which have to store information for every pixel of the graphic.

With the aim of Petri nets model simulation, the textual PNSF3 description must be transformed into another graphic format. In this paper, a conversion into SVG format is proposed. The automatic process of conversion into SVG consists of a few of the following steps:
- validation of the PNSF3 document,
- loading information about Petri nets model;
- changing all the identifiers of net elements (such as places, transitions, arcs, markups) into appropriate SVG graphic elements;
- arranging elements of a Petri graph on the screen;
- assigning colors to appropriate elements;
- generating the script, which is the managing process of animation;
- generating the GUI (graphical user interface), which includes different control panels;
- storing the SVG into a file.

In the previous section it was mentioned that PNSF3 did not keep any information about element arrangement. Therefore, it is very difficult to make an XSL file directly for transformation into SVG graphics. To eliminate these problems, a special program should be used to automate this process.

The first and the most important step for creating the SVG file is to arrange all elements on the screen correctly. Next, the Animate Script is generated on the basis of information about connections, conditions, and results of execution for all transitions. The script consists of a few functions, which control the processes of the animation. Then special functions and control panels are generated. The additional functions enable to communicate with the user. However, the control panels enable setting input signals and watching the current state of the circuit.

The dynamic SVG file consists of three main parts:

- definition block of graphic net elements,
- definition block of control panels, and
- animation script.

5. CONCLUSIONS

In the paper a new XML application for modeling and simulation of digital circuits, especially concurrent controllers, was presented. The proposed PNSF3 is one such textual form for describing Petri net models.

The most important advantages of the proposed PNSF3 are as follows: extreme flexibility, platform independence, precise specification, human reading, easiness of preparation, parsability, transformation, and validation. No special and expensive tools are necessary to prepare documents in PNSF3. However, the disadvantage is difficulty of conversion into other graphic formats. This problem can be solved by a special application, which generates the graphical form of Petri nets as a SVG file on the basis of PNSF3 specification.

On the other hand, for simulation of logic controllers, which are described by Petri nets, HDL-based models are very efficient. In the paper, modeling in Verilog-HDL was presented. Verilog construction *always* has been used for effective description of concurrency. For concurrent place encoding, one-hot method is used; a dedicated flip-flop holds the local state of the controller.

ACKNOWLEDGMENT

The research has been financially supported by the (Polish) Committee of Scientific Research in 2004–2006 (grant 3 T11C 046 26).

REFERENCES

1. E. Best, C. Fernandez, Notations and terminology on Petri net theory, *Petri Net Newsletter*, **23**, 21–46 (1986).
2. T. Kropf, *Introduction to Formal Hardware Verification*. Springer-Verlag, Berlin (1999).
3. R.B. Lyngsø, T. Mailund, Textual interchange format for high-level Petri nets. In: *Workshop on Practical Use of Coloured Petri Nets and Design*. Aarhus University, pp. 47–64 (June 1998).
4. D.E. Thomas, P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston (1998).

5. A. Wegrzyn, P. Bubacz, XML application for modeling and simulation of concurrent controllers. In: *Proceedings of the International Workshop on Discrete-Event Systems Design, DESDes2001*, Przytok, Poland, pp. 215–221 (June 2001).
6. M. Wegrzyn, M. Adamski, J.L. Monteiro, The application of reconfigurable logic to controller design. *Control Engineering Practice, Special Section on Custom Processes, IFAC*, **6**, 879–887 (1998).
7. Extensible Markup Language (XML) 1.0, W3C Recommendation; http://www.w3.org.
8. Scalable Vector Graphics (SVG) 1.0, W3C Candidate Recommendation, http://www.w3.org.

Section III

Synthesis of Concurrent Embedded Control Systems

Chapter 10

OPTIMAL STATE ASSIGNMENT OF SYNCHRONOUS PARALLEL AUTOMATA

Yury Pottosin
National Academy of Sciences of Belarus, Institute of Engineering Cybernetics, Surganov Str., 6, 220012, Minsk, Belarus; e-mail: pott@newman.bas-net.by

Abstract: Three algorithms for assignment of partial states of synchronous parallel automata are considered. Two of them are original; the third one is taken for comparison. One of them is exact; i.e., the number of coding variables obtained by this algorithm is minimal. It is based on covering a nonparallelism graph of partial states by complete bipartite subgraphs. Two other algorithms are heuristic. One of the heuristic algorithms uses the same approach as the exact one. The other is known as iterative. The results of application of these algorithms on some pseudorandom synchronous parallel automata and the method for generating such objects are given.

Key words: synchronous parallel automata; state encoding; parallelism.

1. INTRODUCTION

The parallel automaton is a functional model of a discrete device and is rather convenient to represent the parallelism of interactive branches of controlled process[19]. The main distinction between a parallel automaton and a sequential one (finite state machine) is that the latter can be in only one state at any moment, while the parallel automaton can be in several *partial* states simultaneously. A set of partial states in which a parallel automaton can be simultaneously is called a *total state*. Any two partial states in which an automaton can be simultaneously are called *parallel*.

A parallel automaton is described by the set of strings of the form μ_i : $-w_i \rightarrow v_i \rightarrow \nu_i$, where w_i and v_i are elementary conjunctions of Boolean variables that define the condition of transition and the output signals, respectively, and μ_i and ν_i are labels that represent the sets of partial states of the parallel automaton[19]. Every such string should be understood as follows. If the total state of the parallel automaton contains all the partial states from μ_i and the event w_i has been realized in the input variable space, then the automaton is found to be in the total state that differs from the initial one by containing partial states from ν_i instead of those from μ_i. The values of output variables in this case are set to be such that $\nu_i = 1$.

If $-w_i$ and $\rightarrow \nu_i$ are removed from the string, it can be interpreted as a transition (μ_i, ν_i) in a Petri net. Therefore, the set of such reduced strings can be considered as a Petri net being a "skeleton" of the given parallel automaton. Here we consider only those parallel automata whose skeleton is an α-net[19] that is a subclass of live and safe expanded nets of free choice, which are studied in Ref. 6.

In state assignment of a parallel automaton, partial states are encoded by ternary vectors in the space of introduced internal variables that can take values 0, 1, or "–," orthogonal vectors being assigned to nonparallel states and nonorthogonal vectors to parallel states[2,18,19]. The orthogonality of ternary vectors means existence of a component having opposite values (0 and 1) in these vectors. It is natural to minimize the dimension of the space that results in the minimum of memory elements (flip-flops) in the circuit implementation of the automaton.

The methods to solve the state assignment problem for synchronous parallel automata are surveyed in Ref. 4. Two heuristic algorithms are considered here. One of them is based on iterative method[3]; the other reduces the minimization of the number of memory elements to the problem of covering a nonparallelism graph of partial states by complete bipartite subgraphs[10]. To solve the problem of covering, the algorithm uses a heuristic technique. The third algorithm considered here is exact; i.e., the number of coding variables (memory elements) obtained by this algorithm is minimal. It also finds a cover of a nonparallelism graph of partial states by complete bipartite subgraphs, though using an exact technique[12]. These three algorithms were used to encode partial states of a number of synchronous parallel automata obtained as pseudorandom objects. The pseudorandom parallel automata with given parameters were generated by a special computer program. The method for generating such objects is described. The results of this experiment allow to decide about the quality of the algorithms. Similar experiments are described in Ref. 21, where another approach was investigated and the pseudorandom objects were Boolean matrices interpreted as partial state orthogonality matrices of parallel automata.

2. EXACT ALGORITHM

Below, we refer to this algorithm as Algorithm A. It is based on covering a nonparallelism graph G of partial states by complete bipartite subgraphs.

2.1 Reducing the problem to search for a cover of a graph by complete bipartite subgraphs

A method for partial state assignment that reduces the problem to searching the shortest cover of a graph by complete bipartite subgraphs is known[10,18]. This method considers graph G, whose vertices correspond to partial states of a given automaton and edges to pairs of nonparallel partial states.

In general, graph G is constructed by obtaining all achievable markings of the Petri net that is the skeleton of the given automaton. The number of them grows exponentially with the size of the net. Here the considered automata are restricted to those whose skeletons are α-nets. The characteristic properties of an α-net are the initial marking of it, consisting of one element, $\{1\}$, and the sets of input places of two different transitions coinciding or disjoining. The complexity of the problem of establishing state parallelism relation of such automata is proved to be polynomial[9]. By the way, encoding partial states by ternary vectors is admitted in this case.

According to Ref. 7, a complete bipartite subgraph of a graph $G = (V, E)$ with vertex set V and edge set E is the graph $B = (V', E')$, where $V' \subseteq V$, $E' \subseteq E$, and the vertex set V' is divided into two disjoint subsets X and Y so that $v_1 v_2 \in E'$ if and only if $v_1 \in X$ and $v_2 \in Y$. A cover of graph G by complete bipartite subgraphs is a family of complete bipartite subgraphs, such that every edge of G is at least in one of them. Below, we call it *B-cover* of G. The *shortest B-cover* of G is a B-cover with the minimal number of elements (subgraphs).

Let the complete bipartite subgraphs $B_1, B_2, \ldots, B_m$ form the shortest B-cover of G, and let B_k, $k = 1, 2, \ldots, m$, be associated with a Boolean coding variable z_k so that $z_k = 1$ for the states relative to one partite set of B_k and $z_k = 0$ for the states relative to the other. Then the values of coding variables $z_1, z_2, \ldots, z_m$ represent the solution sought for.

2.2 Decomposition of the partial state nonparallelism graph

The first step to the solution is finding all maximum complete bipartite subgraphs of G. Three ways to do it are given in Refs. 11 and 17. Then one must obtain the shortest cover of edge set of G by those complete bipartite subgraphs.

The search for the shortest cover is a classical NP-hard problem[5]. In our case, the complexity of this problem can be considerably decreased. Let us consider one of the ways to decrease the dimension of complete bipartite subgraph cover problem for the partial state nonparallelism graph G of a parallel automaton. It can be applied if graph G can be represented as $G = G_1 + G_2$, i.e. in the form of the result of join operation on two graphs[7]. If $G = (V, E)$, $G_1 = (V_1, E_1)$, and

$G_2 = (V_2, E_2)$, then $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_{12}$, and E_{12} consists of edges connecting every vertex from V_1 with all vertices from V_2. The decrease is achieved if G_1 or G_2 is a complete graph. This is typical for an automaton whose skeleton is α-net.

Let $\langle X_1, Y_1\rangle$ and $\langle X_2, Y_2\rangle$ be maximum complete bipartite subgraphs of G_1 and G_2, respectively. Then any maximum complete bipartite subgraph of G can be represented in one of the forms $\langle X_1 \cup X_2, Y_1 \cup Y_2\rangle$, $\langle X_1 \cup Y_2, Y_1 \cup X_2\rangle$, $\langle X_1, Y_1 \cup V_2\rangle$, $\langle X_1 \cup V_2, Y_1\rangle$, $\langle X_2 \cup V_1, Y_1\rangle$, $\langle X_2, Y_2 \cup V_1\rangle$, or $\langle V_1, V_2\rangle$. So, having the families of maximum complete bipartite subgraphs of G_1 and G_2, one can easily obtain all maximum complete bipartite subgraphs of G. Let r_1 and r_2 be the numbers of maximum complete bipartite subgraphs of G_1 and G_2, respectively. Then enumerating all the above forms we obtain the number of maximum complete bipartite subgraphs of G as

$$r = 2(r_1 r_2 + r_1 + r_2) + 1. \tag{1}$$

If one of those graphs, namely G_2, is complete graph K_n with n vertices, then any two-block partition of its vertex set defines one of its maximum complete bipartite subgraphs and their number is $r_2 = 2^{n-1} - 1$. The number of maximum complete bipartite subgraphs of G in this case is

$$r = (r_1 + 1)2^n - 1. \tag{2}$$

The class of parallel automata under consideration is featured by exiting at least one partial state nonparallel to any other state. As the initial marking of α-net has only one place, the total initial state of the automaton contains only one partial state that is not parallel to any other partial state.

It can be seen from formula (2) that adding n vertices to a graph, so that each one of them is connected with all other vertices of the graph, increases the number of maximum complete bipartite subgraphs more than 2^n times. We call the vertex that is adjacent to all others an *all-adjacent* vector.

It follows from above that for the search of a shortest B-cover of a partial state nonparallelism graph G, it is convenient to decompose graph G into graphs G_1 and G_2, where G_1 is the subgraph of G, no vertex of which is all adjacent in G, and G_2 is the complete subgraph of G induced by all all-adjacent vertices of G. The idea of using decomposition as the means to reduce the dimension of the task is rather fruitful. For example, one can see in Ref. 8 another case of using decomposition to decrease the dimension of the problem of our field.

2.3 Placement of partial states in Boolean space of coding variables

The classical covering problem is solved now by the traditional way only for G_1. We obtain the B-cover of G_1. It may be considered as a set of m

partial two-block partitions on the vertex set of G. Having introduced Boolean variables $z_1, z_2, \ldots, z_m$ and determined their values for the vertices of G_1 and thus for the corresponding partial states of the given automaton as it is shown above, we obtain the set of ternary vectors that are the codes of those states. These codes form a Boolean space **Z** of coding variables $z_1, z_2, \ldots, z_m$.

Encoding the other vertices of G that are not in G_1 may be described as placement of their codes in the part of Boolean space **Z** that is not occupied by the intervals defined by the above ternary vectors. If these intervals occupy the whole **Z** or the rest of it is not enough to place the codes of all-adjacent vertices of G, then **Z** is widened by adding new variables. The set of the partial state codes form a ternary coding matrix.

It should be noted that the size of the rest of **Z**, where the codes of all-adjacent vertices exist, may be different for different shortest covers. Possibly, it is not the shortest cover that always suits the solution of our problem. Therefore, all nonredundant covers should be considered. The algorithm to find all nonredundant covers can be taken from Ref. 16.

Extracting the part of **Z** that is not covered by the intervals corresponding to the codes of the vertices of G is executed as the solving of the complement problem described in Ref. 17. It is reduced to finding for a given ternary matrix T the ternary matrix U such that any row of U is orthogonal to any row of T and these matrices together specify a Boolean function that is identically equal to 1. As it was said above, the part of **Z** defined by matrix U may not be enough for encoding all all-adjacent vertices of G. In this case, the set of coding variables is extended as much as the space formed by these variables can contain the codes of all vertices of G. The value of every such new variable z_i in the codes of vertices of G_1 is assumed to be 0.

The variant of a nonredundant cover is selected that gives the minimum size of space **Z**.

2.4 Example

Let a parallel automaton be given by the following set of strings:

$$\begin{aligned}
1 &: -\overline{x}_1 x_2 \rightarrow y_1 \overline{y}_2 \rightarrow 10\\
10 &: -\overline{x}_2 \rightarrow 2.3.4\\
2 &: \rightarrow \overline{y}_1 \rightarrow 5\\
3.5 &: -x_2 \rightarrow 8\\
4 &: -\overline{x}_1 \rightarrow \overline{y}_1 \rightarrow 7\\
4 &: -x_1 \rightarrow y_2 \rightarrow 9\\
7 &: -\overline{x}_2 \rightarrow 9\\
8.9 &: \rightarrow \overline{y}_2 \rightarrow 6\\
6 &: -x_1 \rightarrow y_1 \overline{y}_2 \rightarrow 1
\end{aligned}$$

Table 10-1. Obtaining nonredundant covers of graph G_1 by maximum complete bipartite subgraphs

Maximum complete	Edges of G_1							Nonredundant					
bipartite G_1 subgraphs	v_2v_8	v_3v_8	v_2v_5	v_4v_7	v_4v_9	v_5v_8	v_7v_9	covers of G_1					
$\langle v_2,v_8;\, v_5\rangle$			1			1		1	1	1			
$\langle v_2,v_3,v_5;\, v_8\rangle$	1	1				1		1	1	1	1	1	1
$\langle v_4;\, v_7,v_9\rangle$				1	1			1		1	1		1
$\langle v_4,v_9;\, v_7\rangle$				1			1	1	1		1	1	
$\langle v_2;\, v_5,v_8\rangle$	1		1								1	1	1
$\langle v_4,v_7;\, v_9\rangle$					1		1		1	1		1	1

The nonparallelism relation on the set of partial states of this automaton is defined by the following matrix that is the adjacency matrix of graph G:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \end{array}
\begin{array}{c} \begin{array}{cccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \end{array} \\
\left[\begin{array}{cccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}\right] \end{array}$$

Graph G_1 is the subgraph of G induced by the set of vertices $\{v_2, v_3, v_4, v_5, v_7, v_8, v_9\}$, and G_2 is the subgraph induced by $\{v_1, v_6, v_{10}\}$.

The results of the search process of nonredundant covers of graph G_1 by maximum complete bipartite subgraphs are shown in Table 10-1. The rows of Table 10-1 correspond to maximum complete bipartite subgraphs of graph G. The left part of Table 10-1 shows all the maximum complete bipartite subgraphs of G. The middle part is the covering table, its columns corresponding to the edges of G. The columns of the right part of Table 10-1 represent all nonredundant covers of G by its maximum complete bipartite subgraphs, where 1 in a row denotes the presence of the corresponding subgraph in the given cover.

One can construct a ternary matrix for any such cover. All the matrices give sets of intervals, each of which occupies the whole space of four Boolean variables but one element. Therefore, all obtained covers are equivalent. For the cover represented by the fourth column of the right part of Table 10-1, we have the following ternary matrix:

$$\begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} z_1 & z_2 & z_3 & z_4 \end{array} \\ \begin{bmatrix} 1 & - & - & 0 \\ 1 & - & - & - \\ - & 0 & 1 & - \\ 1 & - & - & 1 \\ - & 1 & 0 & - \\ 0 & - & - & 1 \\ - & 1 & 1 & - \end{bmatrix} \end{array}$$

The rows of this matrix are marked by the indices of the partial states of the given automaton, except those having no parallel states. The intervals defined by the rows of the matrix occupy almost the whole space **Z** formed by Boolean variables z_1, z_2, z_3, and z_4. Only one element of it, 0000, is vacant. Therefore, to place the remaining partial states, 1, 6, and 10, in **Z**, it must be widened. Having added variable z_5, we obtain the final coding matrix with the minimum length of codes as follows:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} z_1 & z_2 & z_3 & z_4 & z_5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & - & - & 0 & 0 \\ 1 & - & - & - & 0 \\ - & 0 & 1 & - & 0 \\ 1 & - & - & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ - & 1 & 0 & - & 0 \\ 0 & - & - & 1 & 0 \\ - & 1 & 1 & - & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

3. HEURISTIC ALGORITHMS

The NP-hardness of covering problem[5] does not allow it always to be solved in acceptable time. Therefore the heuristic algorithms that obtain in many cases the shortest cover are developed.

3.1 Algorithm B

The method realized in Algorithm B reduces the problem to covering the state nonparallelism graph G of a given automaton by complete bipartite subgraphs as well as in Algorithm A, but the algorithm for covering is not exact. It consists of two stages. At the first stage the sequence of graphs $G^2, G^3, \ldots, G^n = G$ is considered, where G is the nonparallelism graph of the given automaton with $V = \{v_1, v_2, \ldots, v_n\}$ as the set of vertices, and G^i is the subgraph of G induced by the set of vertices $V^i = \{v_1, v_2, \ldots, v_i\}$. Having the B-cover of G^i the transition from it to the B-cover of G^{i+1} is carried out. At the second stage the obtained B-cover is improved (if possible). This improvement consists in removing some complete bipartite subgraph from the B-cover and in the attempt of reconstruction the B-cover by adding edges to the remaining subgraphs. This procedure repeats for all elements of the B-cover. The complete bipartite subgraphs are obtained concurrently with the constructing of the B-cover.

Let $B^i = \{B_1^i, B_2^i, \ldots, B_{m_i}^i\}$ be the B-cover of graph G^i. When going from G^i to G^{i+1} and adding vertex v_{i+1} and incident edges, we attempt to place these edges in subgraphs $B_1^i, B_2^i, \ldots, B_{m_i}^i$. Evidently, subgraph $B_j^i = \langle X_j^i, Y_j^i \rangle$ can be transformed into some $B' = \langle X_j^i \cup \{v_{i+1}\}, Y_j^i \rangle$ only if $Y_j^i \subseteq N(v_{i+1})$, where $N(v)$ is the neighborhood of vertex v. Some edges connecting v_{i+1} with vertices from $N(v_{i+1})$ would be in B'. The edges not covered can be introduced to other complete bipartite subgraphs in the same way. If we manage to do it for all edges connecting v_{i+1} with vertices from V^i, then $|B^{i+1}| = m_{i+1} = |B^i| = m_i$. If not, then $m_{i+1} = m_i + 1$ and $\langle X_{m_{i+1}}^{i+1}, Y_{m_{i+1}}^{i+1} \rangle$ is taken as $B_{m_{i+1}}^{i+1}$, where $X_{m_{i+1}}^{i+1} = \{v_{i+1}\}$ and $Y_{m_{i+1}}^{i+1}$ consists of those vertices from V^{i+1} that are connected with v_{i+1} by the edges that are not in any of $B_1^{i+1}, B_2^{i+1}, \ldots, B_{m_i}^{i+1}$.

The initial B-cover may consist of a single edge (being in G^2), or it may be a star.

The size of a B-cover obtained in such a way depends to a great extent on the numbering of vertices in a given graph. The results of solving test tasks allow assuming that the most favorable numbering is that obtained as follows. The vertex v with the maximum degree in graph G gets number n. After removing v with incident edges we take the vertex with the maximum degree in graph $G - v$. We associate number $n-1$ with it and remove it as well. This process of þlucking" G continues until only two vertices remain in it. We associate with them arbitrary numbers 1 and 2 and other vertices are numbered in the order reverse to the course of the process.

After a B-cover of graph G has been obtained in such a way, we attempt to decrease the number of elements of the B-cover. The possibility of such decreasing is due to the way of constructing a B-cover.

We improve the obtained B-cover in the following way. Let us have a B-cover $B_1, B_2, \ldots, B_m$ of a graph $G = (V, E)$. For this B-cover, we call an edge $e \in E$ *once-covered* if there is only one B_i that has e. For every B_i, $i = 1, 2, \ldots, m$, we calculate the number s_i of once-covered edges belonging to it. We remove from the B-cover the subgraph B_j whose s_j is minimal. Then we try to restore the B-cover by adding vertices from V and edges from E to subgraphs of the rest. If we cannot manage it we introduce a new complete bipartite subgraph containing all uncovered edges.

This procedure can be repeated many times. The sign to finish it may be the following condition: the number m of B-cover elements and the number of once-covered edges do not decrease. When this condition is satisfied the process of solving the task is over and the B-cover obtained after the last executing the procedure is the result.

To illustrate Algorithm B, let us take the parallel automaton from section 2.4. The adjacency matrix of the state nonparallelism graph of this automaton is given above. After renumbering according to the rule above, it is

$$\begin{array}{c|cccccccccc|}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \\
v_1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
v_2 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
v_3 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
v_4 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
v_5 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
v_6 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
v_7 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
v_8 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
v_9 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
v_{10} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{array},$$

which corresponds to the following order of vertices: v_3, v_5, v_9, v_7, v_2, v_4, v_8, v_{10}, v_6, v_1.

The sequence of B-covers corresponding to the sequence of subgraphs $G^2, G^3, \ldots, G^{10}$ of graph G is

$B^2 = B^3 = \varnothing$;

$B^4 = \langle v_3; v_4\rangle$;

$B^5 = \langle v_3; v_4\rangle; \langle v_2; v_5\rangle$;

$B^6 = \langle v_3, v_6; v_4\rangle; \langle v_2; v_5\rangle; \langle v_3; v_6\rangle$;

$B^7 = \langle v_3, v_6; v_4\rangle; \langle v_2, v_7; v_5\rangle; \langle v_3; v_6\rangle; \langle v_1, v_2; v_7\rangle$;

$B^8 = \langle v_3, v_6; v_4, v_8\rangle; \langle v_2, v_7; v_5, v_8\rangle; \langle v_3, v_8; v_6\rangle; \langle v_1, v_2; v_7, v_8\rangle;$
$\langle v_4, v_5; v_8\rangle$;

$B^9 = \langle v_3, v_6, v_9; v_4, v_8\rangle; \langle v_2, v_7; v_5, v_8, v_9\rangle; \langle v_3, v_8, v_9; v_6\rangle;$
$\langle v_1, v_2; v_7, v_8, v_9\rangle; \langle v_4, v_5; v_8, v_9\rangle; \langle v_3; v_9\rangle$;

$B^{10} = \langle v_3, v_6, v_9; v_4, v_8, v_{10}\rangle; \langle v_2, v_7, v_{10}; v_5, v_8, v_9\rangle; \langle v_3, v_8, v_9; v_6\rangle;$
$\langle v_1, v_2; v_7, v_8, v_9, v_{10}\rangle; \langle v_4, v_5; v_8, v_9, v_{10}\rangle; \langle v_3; v_9\rangle; \langle v_7; v_{10}\rangle.$

Let us try to decrease the number of elements of the obtained B-cover. Subgraph $B_6^{10} = \langle v_3; v_9 \rangle$, as well as $B_7^{10} = \langle v_7; v_{10} \rangle$, has the minimum number of once-covered edges ($s_6 = s_7 = 1$). We remove B_6^{10} from B^{10} and extend $\langle v_4, v_5;\ v_8, v_9, v_{10} \rangle$ to $\langle v_3, v_4, v_5; v_8, v_9, v_{10} \rangle$. Then we remove B_7^{10} and extend $\langle v_3, v_4, v_5; v_8, v_9, v_{10} \rangle$. Finally, we obtain the following B-cover: $\langle v_3, v_6, v_9; v_4, v_8, v_{10} \rangle$; $\langle v_2, v_7, v_{10}; v_5, v_8, v_9 \rangle$; $\langle v_3, v_8, v_9; v_6 \rangle$; $\langle v_1, v_2; v_7, v_8, v_9, v_{10} \rangle$; $\langle v_3, v_4, v_5, v_7; v_8, v_9, v_{10} \rangle$.

3.2 Algorithm C

To appreciate the efficiency of the proposed algorithms we consider Algorithm C based on the heuristic iterative method suggested in Ref. 3. The iterative method assumes the definition of parallelism relation and an initial coding matrix for partial states (the initial matrix may be empty). The matrix is extended in the process of coding by introducing additional coding variables, which makes it possible to separate nonparallel partial states in certain pairs. To separate two states means to put opposite values (0 and 1) to some coding variable in the codes of these states. The method consists in iterative executions of two procedures: introducing a new coding variable and defining its values in codes of nonseparated yet non-parallel partial states. These procedures are executed until all nonparallel states have been separated. Minimizing the number of introduced coding variables also minimizes the Hamming distance between codes of states related by transitions. The aim of this is the minimization of the number of switchings of RS-type flip-flops in circuit realization of a parallel automaton.

Introducing a new coding variable is accompanied with separating the maximal number of nonseparated yet nonparallel partial states by this variable. For this purpose, at each step of the procedure of defining the values of the due variable, a state is chosen to encode by this variable. This state should be separated from the maximal number of states already encoded by this variable. The number of states that are not separated from the chosen one and have been encoded by this variable must be maximal. A new coding variable is introduced if the inner variables already introduced do not separate all nonparallel partial states from each other.

4. GENERATING PARALLEL AUTOMATA

Any string of the form $\mu_i : -w_i \to v_i \to v_i$ in automaton specification we call a *transition*, and a set of transitions with the same μ_i is a *sentence*. The algorithm for generating parallel automata is described in detail in Ref. 13, where a parallel automaton is constructed as a system of three pseudorandom objects. They are the "skeleton" of the automaton that is an α-net specified in the form of a sequence of pairs (μ_i, v_i), the ternary matrix ***X*** representing conjunctions w_i, and the ternary matrix ***Y*** representing conjunctions v_i. In our task the α-net is enough; therefore, we should not describe the way of generating ***X*** and ***Y*** here.

The parameters given beforehand of every pseudorandom α-net generated by a special computer program are the number of places (partial states of the automaton) p, the number of transitions t, and the number of sentences s.

Generating pseudorandom parallel automata as systems of three objects mentioned above with parameters given beforehand would not be difficult if no correctness demands exist, without which there is no sense in executing algorithms intended for such automata. Proceeding from the correctness properties of a parallel control algorithm that are named in Ref. 19, let us consider the following properties of a parallel automaton that guarantee its correctness in our case. It must be irredundant (there is no transition that can be never executed), recoverable (it can return to the initial total state from any other one), and self-coordinated (any transition cannot be started again before it ceases).

Irredundancy, recoverability, and self-coordination of a parallel automaton correspond to liveness and safeness of the related α-net[19]. In the Petri net theory the reduction methods for checking liveness and safeness are well known[1], where the initial net is transformed according to certain rules preserving these properties. The transformations reduce the dimension of a given net and so facilitate checking the liveness and safeness of the net.

To check liveness and safeness of α-nets the application of two rules is sufficient[19]. The first rule consists in deleting loops, i.e., the transitions where $\mu_i = v_i$. The second is as follows. Let a set of places π not containing place 1 be such that for every transition (μ_i, v_i), $\pi \cap \mu_i \neq \varnothing$ implies $\pi = \mu_i$ and $\pi \cap v_i = \varnothing$, and $\pi \cap v_i \neq \varnothing$ implies $\pi \subseteq v_i$. Besides, there exists at least one transition with $\pi \cap v_i \neq \varnothing$. Then all transitions ($\mu_j$, v_j) with $\pi = \mu_i$ are removed and every transition (μ_k, v_k) with $\pi \subseteq v_k$ is substituted by the set of transitions that are obtained from (μ_k, v_k) by replacing π by sets v_j from those transitions (μ_j, v_j) where $\pi = \mu_j$. A live and safe α-net is proved in Ref. 15 to be completely reducible; i.e., the application of these rules leads to the net that consists of the only transition (1, 1). This implies the way of generating live and safe α-nets, which consists in transformations that are inverse to the above.

5. EXPERIMENTAL RESULTS

Algorithms A, B, and C are realized in computer programs and the corresponding modules are included as components into ISAPR, which is a research CAD system[14]. The program for generating pseudorandom parallel automata is

Table 10-2. Experimental results (p, t, and s are parameters of α-nets, b is the number of maximum complete bipartite subgraphs of G_2)

					Algorithm A		Algorithm B		Algorithm C	
Name	p	t	s	B	Code length	Run time	Code length	Run time	Code length	Run time
AP2	20	18	18	75	6	13 min. 28 sec.	7	6 sec.	6	3 sec.
APR1	2	2	1	8	5	8 sec.	6	7 sec.	5	3 sec.
APR2	2	2	1	4	5	5 sec.	6	8 sec.	5	3 sec.
APR3	2	2	1	7	4	6 sec.	5	3 sec.	6	3 sec.
APR6	2	2	1	43	5	2 min. 23 sec.	6	8 sec.	6	3 sec.
APR7	2	3	1	55	5	49 sec.	6	8 sec.	6	3 sec.
APR8	2	1	1	49	5	1 min. 28 sec.	5	5 sec.	5	3 sec.
RAZ	2	2	1	1033	9	3 h. 46 min. 22 sec.	9	8 sec.	10	4 sec.

included into ISAPR as well. This program was used to generate several parallel automata. The results of partial state assignment are shown in Table 10-2. One of the automata whose partial states were encoded, RAZ, was not generated by the program mentioned above. It was obtained from a real control algorithm.

As was noted, only the parameters of α-net, i.e., the number of places p, the number of transitions t, and the number of sentences s, were considered. Besides this, the number of maximum complete bipartite subgraphs in the graph G of nonparallelism of partial states of the given automaton may be of interest. Algorithm A uses the method that decomposes graph G into two subgraphs G_1 and G_2, G_1 being complete. The maximum complete bipartite subgraphs were found in G_2. The calculations were performed on a computer of AT type with the 386 processor.

6. CONCLUSION

The technique of investigation of algorithms for state assignment of parallel automata is described in this paper. The experimental data show that Algorithms B and C are quite competitive to each other, although the speed of Algorithm C is higher than that of Algorithm B. Algorithm A is intended to be applied for automata of small dimension. It can be used as a standard algorithm and helps one to appreciate the quality of solutions obtained by heuristic algorithms.

REFERENCES

1. S.M. Achasova, O.L. Bandman, *Correctness of Concurrent Computing Processes*. Nauka, Siberian Division, Novosibirsk (1990) (in Russian).

2. M. Adamski, M. Wegrzyn, Field programmable implementation of current state machine. In: *Proceedings of the Third International Conference on Computer-Aided Design of Discrete Devices (CAD DD'99)*, Vol. 1, Institute of Engineering Cybernetics, National Academy of Sciences of Belarus, Minsk, pp. 4–12 (1999).
3. L.D. Cheremisinova, State assignment for parallel synchronous automata. *Izvestia AN BSSR, Ser. Fisiko-Tehnicheskih Nauk*, **1** 86–91 (1987) (in Russian).
4. L.D. Cheremisinova, Yu.V. Pottosin, Assignment of partial states of a parallel synchronous automaton. In: *Proceedings of the International Conference on Computer-Aided Design of Discrete Devices (CAD DD'95)*. Wydawnictwo Uczelniane Politechniki Szczecinskiej, Minsk-Szczecin, pp. 85–88 (1995).
5. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory on NP-Completeness*. W.M. Freeman & Company, San Francisco, CA (1979).
6. M.T. Hack, Analysis of production schemata by Petri nets. Project MAC TR-94, Cambridge (1972).
7. F. Harary, *Graph Theory*. Addison-Wesley Publishing Company, Reading, MA (1969).
8. A. Karatkevich, Hierarchical decomposition of safe Petri nets. In: *Proceedings of the Third International Conference on Computer-Aided Design of Discrete Devices (CAD DD'99)*, Vol. 1. Institute of Engineering Cybernetics, National Academy of Sciences of Belarus, Minsk, pp. 34–39 (1999).
9. A.V. Kovalyov, Concurrency relation and the safety problem for Petri nets. In: *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, Sheffild, pp. 299–309 (1992).
10. Yu.V. Pottosin, Covering a graph by complete bipartite subgraphs. In: *Design of Discrete Systems*. Institute of Engineering Cybernetics of Academy of Sciences of Belarus, Minsk, pp. 72–84 (1989) (in Russian).
11. Yu.V. Pottosin, Finding maximum complete bipartite subgraphs in a graph. In: *Automatization of Logical Design of Discrete Systems*. Institute of Engineering Cybernetics of Academy of Sciences of Belarus, Minsk, pp. 19–27 (1991) (in Russian).
12. Yu.V. Pottosin, A method for encoding the partial states of a parallel automaton with minimal-length codes. In: *Automatic Control and Computer Sciences*, Vol. 6. Allerton Press, Inc., New York, pp. 45–50 (1995).
13. Yu.V. Pottosin, Generating parallel automata. In: *Methods and Algorithms for Logical Design*. Institute of Engineering Cybernetics of Academy of Sciences of Belarus, Minsk, pp. 132–142 (1995) (in Russian).
14. N.R. Toropov, *Research CAD System for Discrete Control Devices: Materials on Software for Computers*. Institute of Engineering Cybernetics of Academy of Sciences of Belarus, Minsk (1994) (in Russian).
15. V.V. Tropashko, Proof of the conjecture of complete reducibility of α-nets. In: *Design of Logical Control Systems*. Institute of Engineering Cybernetics of Academy of Sciences of BSSR, Minsk, pp. 13–21 (1986) (in Russian).
16. A.D. Zakrevskij, Optimal coverage of sets. In: *A Programming Language for Logic and Coding Algorithms*. ASM Monograph Series, Academic Press, New York, London, pp. 175–192 (1969).
17. A.D. Zakrevskij, Combinatorics of logical design. *Avtomatika i Vychislitelnaya Tekhnika*, **2** 68–79 (1990) (in Russian).
18. A.D. Zakrevskij, Optimization of matrix of partial state assignment for parallel automata. In: *Formalization and Automatization of Logic Design*. Institute of Engineering Cybernetics, Academy of Sciences of Belarus, Minsk, pp. 63–70 (1993) (in Russian).
19. A.D. Zakrevskij, Parallel logical control algorithms: verification and hardware implementation. *Computer Science Journal of Moldova*, **4** (1), 3–19 (1996).

20. A.D. Zakrevskij, *Parallel Algorithms for Logical Control*. Institute of Engineering Cybernetics, National Academy of Sciences of Belarus, Minsk (1999) (in Russian).
21. A. Zakrevskij, I. Vasilkova, A quick algorithm for state assignment in parallel automata. In: *Proceedings of the Third International Conference on Computer-Aided Design of Discrete Devices (CAD DD'99)*, Vol. 1. Institute of Engineering Cybernetics, National Academy of Sciences of Belarus, Minsk, pp. 4044 (1999).

Chapter 11

OPTIMAL STATE ASSIGNMENT OF ASYNCHRONOUS PARALLEL AUTOMATA

Ljudmila Cheremisinova
Institute of Engineering Cybernetics of National Academy of Sciences of Belarus, Surganov Str., 6, 220012 Minsk, Belarus; e-mail: cld@newman.bas-net.by

Abstract: A problem of race-free state assignment of asynchronous parallel automata is considered. The goal is to encode partial states of parallel automaton using minimal number of coding variables and excluding critical races during automaton operation. Requirements imposing on the partial states codes to eliminate the inflence of races are formulated. An exact algorithm to find a minimal solution of the problem of race-free state assignment for parallel automata is suggested. The algorithm provides reducing the computational efforts when searching for state encoding.

Key words: asynchronous parallel automata; state assignment; parallelism; critical races.

1. INTRODUCTION

Successive control of a multicomponent system depends greatly on the efficiency of the synchronization among its processing elements. The functions of a control of such a system are concentrated in one block – a logic control device that should provide a proper synchronization of interaction between the components. In order to represent clearly the interaction involved in concurrent engineering system, it is necessary to describe formally its functional and structural properties.

As a functional model of a discrete control device to be designed, a model of parallel automaton is proposed[1,8,9]. This model can be considered as an extension of a sequential automaton (finite state machine) allowing representing parallel processes. The parallel automaton is a more complicated and less studied model in contrast with the classical sequential automaton model. An essential difference from sequential automaton is that a parallel automaton can be in more than one state simultaneously. That is why the states of a parallel automaton were called *partial*[8]. Partial states in which a parallel automaton is at the same moment are called *parallel*[8]. Any transition of automaton defines parallel partial state changes.

The design of asynchronous automata has been an active area of research for the last 40 years. There has been a renewed interest in asynchronous design because of its potential for high performance. However, design of asynchronous automata remains a cumbersome problem because of difficulties of ensuring correct dynamic behavior.

The important step of control device hardware implementation is the state assignment. It is at the heart of the automaton synthesis problem (especially for its asynchronous mode of realization). Despite great efforts devoted to this problem, no satisfactory solutions have been proposed. A difference of this process for parallel automaton in comparison with the sequential one is that there are parallel states (they are compatible in the sense that the automaton can find itself in several of them at the same time). That is why it was suggested in Ref. 8 to encode partial states with ternary vectors that should be nonorthogonal for parallel partial states but orthogonal for nonparallel ones. In such a way an initial parallel automaton is transformed from its abstract form into a structural form -a sequent parallel automaton [9] or a system of Boolean functions that can be directly implemented in hardware.

The problem of state assignment becomes harder when asynchronous implementation of a parallel automaton is considered. The mentioned condition imposed on codes is necessary but not enough for that case. The additional condition to be fulfilled is to avoid the inflence of races between memory elements (flp-flps) during hardware operation. One of the ways to avoid this is to order switches of memory elements so as to eliminate critical races.

A problem of racefree state assignment of asynchronous parallel automata is considered. The goal is to encode partial states of parallel automaton using minimal number of coding variables and to avoid the critical races during automaton operation. An exact algorithm to find a minimal solution of the problem is suggested. The algorithm allows reducing computational efforts of state encoding. The same problem is considered in Ref. 5, where another approach is suggested. The method is based on covering a family of complete bipartite subgraphs defining constraints of absence of critical races by minimal number of maximal complete bipartite subgraphs of the partial state nonparallelism graph.

2. RACE-FREE IMPLEMENTATION OF ASYNCHRONOUS AUTOMATON

The asynchronous sequential automaton behaves as follows. Initially, the automaton is stable in some state. After the input state changes, the output signals change their values as specified in automaton description. An internal state change may be concurrent with the output change. When the automaton has achieved a new stable state it is ready to receive the next input. Throughout this cycle, output and inner variables should be free of glitches. In summary, asynchronous designs differ from those synchronous because state changes may pass through intermediate states.

The sequence of these intermediate states must be preserved in the case of multi-output change (when intermediate states involve the output change). It can be done with the proper state assignment. The one~~h~~ot encoding [4] can ensure such a behavior, but it demands too many coding variables. That is why the methods of race-free state assignment are of interest.

2.1 Constraints of race-free implementation of sequential automaton

In Ref. 6 the constraints to ensure hardware implementation of sequential automaton to be race-free are given. These constraints allow avoiding interference between automaton transitions that take place for the same input state. Codes satisfying these constraints ensure race-free implementation of the automaton. The encoding constraints can be represented in the form of dichotomies.

A dichotomy is a bipartition $\{S_1; S_2\}$ of a subset of $S : S_1 \cup S_2 \subseteq S$, $S_1 \cap S_2 = \varnothing$. In the state encoding considered, a binary variable y_i covers dichotomy $\{S_1; S_2\}$ if $z_i = 0$ for every state in S_1 and $z_i = 1$ for every state in S_2 (or vice versa). Transitions taking place at the same input are called *competing transitions*. In Ref. 6 the following constraints of critical race-free encoding are given that are induced by pairs of competing transitions of different types:

1. $s_i \to s_j, s_k \to s_l$ $(i, j, k, l$ are pairwise different) give rise to $\{s_i\ s_j;\ s_k,\ s_l\}$;
2. $s_i \to s_j, s_j \to s_l$ $(i, j, l$ are pairwise different) give rise to $\{s_i\ s_j;\ s_l\}$; and $\{s_i; s_j, s_l\}$;
3. $s_i \to s_j, s_k \to s_l$ $(i, j, k,$ are pairwise different) give rise to $\{s_i\ s_j;\ s_l\}$ if the output on the transition from s_k is different from that on the transitions from s_i and s_j (at the input considered).

2.2 Distinctive features of parallel automaton

All parallel partial states in which a parallel automaton is at some moment form its global state. Any transition of automaton defines the partial state changes that cause the global state changes. Most transitions (and all for asynchronous parallel automaton) are forced by changes of external signals.

A parallel automaton is described by a set of generalized transitions $(X_{kl}, S_k) \to (S_l, Y_{kl})$ between the subsets of partial states. Such a transition should be understood as follows: if a global state of the parallel automaton contains all the partial states from S_k and the variables in the conjunction term X_{kl} assume values at which $X_{kl} = 1$, then as the result of the transition the automaton goes to the next global state that differs from initial one in that it contains partial states from S_l instead of those from S_k. More than one generalized transition may take place at some moment when a parallel automaton functions. These transitions define changing different subsets of parallel partial states. There are no races on such a pair of transitions.

In the case of parallel automaton there are generalized transitions instead of elementary ones. A generalized transition $t_{kl} : S_k \to S_l$ consists of $|S_k| \cdot |S_l|$ elementary transitions $s_{ki} \to s_{lj}$, where $s_{ki} \in S_k$ is nonparallel to $s_{lj} \in S_l$. Let us introduce the set $T(t_{kl}, t_{pq})$ of pairs of elementary transitions $s_{ki} \to s_{lj}$ and $s_{pi} \to s_{qj}$ between pairwise nonparallel partial states taken from S_k, S_l, S_p, and S_q generated by the pair of competing transitions t_{kl}: $S_k \to S_l$ and t_{pq}: $S_p \to S_q$. For compatible pair t_{kl}, t_{pq} of generalized transitions, we have $T(t_{kl}, t_{pq}) = \varnothing$.

2.3 Constraints of race-free implementation of parallel automaton

In Ref. 2, it has been shown that in order to avoid the inflence of races on competing generalized transitions t_{kl} and t_{pq}, it is sufficient to avoid them on one pair of elementary transitions from the set $T(t_{kl}, t_{pq})$. This statement gives the way of a parallel automaton partial states encoding. Besides, this statement ensures that any dichotomy constraint consists of pairwise nonparallel partial states. The last implies the absence of a constraint forcing a coding variable to have orthogonal values in codes of parallel partial states.

Let us distinguish elementary u_{nij}, simple u^p_{ni}, and generalized U_n constraints. The first is a single dichotomy constraint. The second is associated with a pair of competing elementary transitions and can consist of one (cases 1 and 3 of constraints) or two (case 2) elementary constraints. To avoid critical races on a pair of elementary competing transitions, one has to satisfy an appropriate simple constraint (one or two elementary ones). A generalized constraint U_n induced by a pair P_n of competing generalized transitions consists of the simple constraints induced by pairs of elementary transitions from its generated set $T(P_n)$. To avoid critical races on P_n it is sufficient to satisfy one of the simple constraints from U_n.

Example 1. Let us consider the following parallel automaton in the form

X_{kl}		$S_k \to S_l$	Y_{kl} :
t_1:	$'x_1$	$s_1 \to s_2 \cdot s_3$	$y_1 y_2$
t_2:	$'x_2 x_3$	$s_2 \to s_9$	$'y_2 y_3$
t_3:	$'x_3$	$s_9 \to s_2$	$y_2' y_3$
t_4:	x_2	$s_2 \to s_4 \cdot s_5$	$'y_1 y_3$

t_5:	x_3	$s_3 \rightarrow s_6$	y_4
t_6:	$x_1{}'x_2$	$s_4 \rightarrow s_7$	$y_1{}'y_2$
t_7:	$'x_2x_3$	$s_5 \rightarrow s_8$	$'y_3$
t_8:	$'x_3$	$s_6 \cdot s_7 \cdot s_8 \rightarrow s_1$	$'y_1{}'y_4$

The partial states from $\{s_2, s_4, s_5, s_7, s_8, s_9\}$ and $\{s_3, s_6\}$ are pairwise parallel, so also are the partial states from $\{s_4, s_7\}$ and $\{s_5, s_8\}$. One can see, for example, that the pair t_1, t_8 of generalized transitions is competing. The generalized constraint $U_{1,8}$ induced by that pair consists of three simple constraints: $u^p_{1,8.1} = (\{s_1, s_2; s_7\}$ and $(\{s_1, s_7; s_2\})$, $u^p_{1,8.2} = (\{s_1, s_2; s_8\}$ and $(\{s_1, s_8; s_2\})$, and $u^p_{1,8.3} = (\{s_1, s_3; s_6\}$ and $(\{s_1; s_3, s_6\})$. Thus for this automaton we have the following set of generalized constraints U_k (in the form of dichotomies) derived from the pairs of competing generalized transitions:

1. $\{s_1, s_2; s_9\}$ and $\{s_1; s_2, s_9\}$;
2. $(\{s_1, s_2; s_4\}$ and $\{s_1; s_2, s_4\})$ or $(\{s_1, s_2; s_5\}$ and $\{s_1; s_2, s_5\})$;
3. $(\{s_1, s_3; s_6\}$ and $\{s_1; s_3, s_6\})$;
4. $\{s_1, s_2; s_5, s_8\}$;
5. $(\{s_1, s_2; s_7\}$ and $\{s_1, s_7; s_2\})$ or $(\{s_1, s_2; s_8\}$ and $\{s_1, s_8; s_2\})$ or $(\{s_1, s_3; s_6\}$ and $\{s_1, s_6; s_3\})$;
6. $\{s_2, s_9; s_4, s_7\}$;
7. $(\{s_2, s_9; s_4\}$ and $\{s_4, s_2; s_9\})$ or $(\{s_2, s_9; s_5\}$ and $\{s_2, s_5; s_9\})$;
8. $\{s_1, s_7; s_2, s_9\}$ or $\{s_1, s_8; s_2, s_9\}$;
9. $\{s_2, s_9; s_5, s_8\}$;
10. $\{s_1, s_7; s_2, s_4\}$ or $\{s_1, s_8; s_2, s_5\}$;
11. $(\{s_1, s_7; s_4\}$ and $\{s_1; s_4, s_7\})$.

Just as in the case of sequential automaton[7], when seeking critical race-free partial state assignment we proceed in three steps:

1. Generating a set of encoding constraints
2. Finding a compressed set of encoding constraints equivalent to the initial one
3. Solving these constraints to produce a partial state assignment.

3. GENERATING AND COMPRESSING A SET OF ENCODING CONSTRAINTS

Now an encoding problem formulation is presented that is based on a matrix notation similar to that used in Ref. 7 for sequential automata. A dichotomy constraint $\{s_i, s_j; s_k, s_l\}$ can be presented as a ternary (3-valued) vector called a constraint vector. Its length equals the number of partial states, ith and jth entries are 1, kth and lth entries are 0 (or vice versa), and the other ones are

“–”(don't care). For example, the dichotomy $\{s_1, s_7; s_2, s_9\}$ corresponds to the vector “1 0 – – – – – 1 – 0.”

Constraint matrix $\boldsymbol{U}$ is a ternary matrix with as many rows as critical race-free constraints exist (for a given automaton) and as many columns as partial states. One can see that the constraint matrix $\boldsymbol{U}$ is an encoding matrix $\boldsymbol{V}$ looked for, but the number of encoding variables (which is equal to the number of rows of $\boldsymbol{U}$) is too large. The task is to compress the matrix $\boldsymbol{U}$ without violation of any constraint from $\boldsymbol{U}$: each row of $\boldsymbol{U}$ should be implicated by some row from $\boldsymbol{V}$. For the case of sequent automaton it is enough. But a parallel automaton constraint matrix $\boldsymbol{U}$ has a complex structure – it consists of submatrices $\boldsymbol{U}_i$ defining generalized constraints U_i; the last ones are in turn 1 or 2 line sectioned (separating simple constraints).

3.1 Relations on the set of constraints

Below, some definitions follow; note that the ternary vectors in every definition are of the same length. A ternary vector $\boldsymbol{a}$ covers a ternary vector $\boldsymbol{b}$ if, whenever the ith entry of $\boldsymbol{b}$ is $\sigma \in \{1,0\}$, the ith entry of $\boldsymbol{a}$ is σ too. $\boldsymbol{b}$ is an inversion of $\boldsymbol{a}(b = {}'a)$ if, whenever the ith entry of $\boldsymbol{a}$ is 1, 0, “–”, the ith entry of $\boldsymbol{b}$ is 0, 1, “–”, respectively. Any vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are orthogonal if for at least an index i the ith entries of $\boldsymbol{a}$ and $\boldsymbol{b}$ are orthogonal (1 and 0, or vice versa).

An elementary constraint u_j implicates an elementary constraint u_i if u_j as a ternary vector covers u_i or its inversion. An elementary constraint u_j can implicate a simple constraint u_n^p if it consists of the only elementary constraint that is implicated by u_j.

A simple constraint u_n^p implicates

- an elementary constraint u_j if u_j is implicated by one of the elementary constraints from u_n^p,
- a simple constraint u_m^p if every $u_{mj} \in u_m^p$ is implicated by at least one of $u_{nj} \in u_n^p$,
- a generalized constraint U_k if u_n^p implicates at least one of $u_{ki}^p \in U_k$.

A generalized constraint U_k implicates

- a simple constraint u_j^p (elementary constraint u_j) if every $u_{kj}^p \in U_k$ implicates it,
- a generalized constraint U_n if every $u_{kj}^p \in U_k$ implicates at least one of $u_{ni}^p \in U_n$.

A set U' of elementary constraints implicates a simple constraint u_{kj}^p and thus a generalized constraint U_k (such that $u_{kj}^p \in U_k$) if every $u_{kj,i} \in u_{kj}^p$ is implicated by one of the constraints from U'.

3.2 Compressing a set of encoding constraints

For computational efficiency of the procedure of searching for an optimal encoding, it is important to reduce the number of rows of constraint matrix $\boldsymbol{U}$ to the minimal number that represents an equivalent set of constraints on the encoding. It is trivial that duplicate generalized constraints can be deleted. Then the number of rows of $\boldsymbol{U}$ can be decreased further by eliminating generalized, simple, or elementary constraints that are implicated by any other generalized constraint.

Example 2. For the automaton under consideration we can see that the generalized constraint $(\{s_1, s_7; s_2, s_9\}$ or $(\{s_1, s_8; s_2, s_9\})$ induced by the pair t_3, t_8 of competing transitions implicates the elementary constraint $(\{s_1; s_2, s_9\}$ from the simple constraint $(\{s_1, s_2; s_9\}$ and $(\{s_1; s_2, s_9\})$ induced by the pair t_1, t_2 of competing transitions.

The irredundant constraint matrix $\boldsymbol{U}$ for the automaton under consideration is shown in the second column in Table 11-1. Its first column gives the structure

Table 11-1. Encoding constraints, their boundary vectors, and compatibility relation among them

Constraint numeration	Constraints 12345 6789	Boundary vectors 12345 6789	Compatibility relation 12345 67890123456789012
1 –1,1	11−−−−−−− 0	11+ −− + −−0	−
2 –2,1.1	11−0−−−−−−	11+00+−0−	1−
3 –2,1.2	10−0−−−−−−	10+00+−0−	0−−
4 –2,2	10−−0−−−−	10+00+0−−	0−−−
5 –3,1.1	1−1−−0−− −	1+1++0+ + +	0000−
6 –3,1.2	1−0−−0−−−	100000000	0011−−
7 –4,1	11−−0−−0−−	11+00+00−	110000−
8 –5,1.1	11−−−− 0−−	11+−0+00+	1100001−
9 –5,1.2	10−−−− 1−−	10+−1+11+	0010000−−
10 –5,2	10−−−−−− 1−	10+1−+11−	0001000−−−
11 –5,3	1−0−−1−−−	1+0++1+ + +	0000000−−−−
12 –6,1	−1−0−−0−1	−1+00+001	01000011110−
13 –7,1	−1−1−−−−0	−1+11+−10	101100011000−
14 –7,2	−1−−1−−−0	−1+11+1−0	101100000100−−
15 –8,1	10−−−− 1−0	10+−1+110	00100000110100−
16 –8,2	10−−−−−− 10	10+1−+110	00010000110100−−
17 –9,1	−1−−0−−01	−1+00+001	0100001111010011−
18 –10,1	10−0−−1−−	10+0++1+−	00100000100010100−
19 –10,2	10−−0−−1−	10++0++1−	00010000010001010−−
20 –11,1.1	1−−0−−1−−	1−+0++1+−	1110000010001010011−
21 –11,1.2	1−−0−−0−−	1−000000−	1111011100011100100−−
22 –12,1	−−−−1−−0−	− − ++1++0−	1001000000100010100100−

of matrix $\boldsymbol{U}$ –what rows belong to what generalized and simple constraints. The last row of the matrix is introduced since nonparallel partial states should be encoded with orthogonal codes (but appropriate constraint is implicated by none other).

The encoding matrix $\boldsymbol{V}$ is grown from an initial seed constraint matrix $\boldsymbol{U}$ by its compressing at the expense of combining some constraints and substituting them for one constraint implicating them.

3.3 Basic definitions

Now we give some definitions and derive some useful properties from them. A constraint u (having no orthogonal entries associated with parallel states) is called an implicant of a set of rows of constrained matrix $\boldsymbol{U}$ if it implicates each of them separately. A set of elementary constraints is considered compatible if there exists an implicant for it. For example, $u_{1,1.1} \in U_1$ is compatible with only $u_{4,1.1} \in U_4$ (their implicants are $\{s_1, s_2; s_5, s_8, s_9\}$, $\{s_1, s_2; s_4, s_5, s_8, s_9\}$, $\{s_1, s_2; s_4, s_5, s_7, s_8, s_9\}$), but not compatible with $u_{10,1.1} \in u^p_{10,1} \in U_{10}$.

Let us define for a row $\boldsymbol{u}_k \in U$ (specifying dichotomy $\{S_{k1}; S_{k2}\}$) a boundary vector $\boldsymbol{u}^z_k$ a 4-valued vector that gives an upper bound of its growth (extension), i.e., it determines the potential of defining the components of the row $\boldsymbol{u}_k$ (considered to be a vector). Its ith entry is σ if the ith entry of $\boldsymbol{u}_k$ is $\sigma \in \{1, 0\}$; otherwise its value depends on the partial state s_i associated with the entry:

1. if it is nonparallel to any of the states from $S_{k1} \cup S_{k2}$, then the entry is “–”;
2. if it is parallel only to those states from $S_{k1} \cup S_{k2}$ and the appropriate entries of u_i have the same value $\delta \in \{1, 0\}$, then the entry is δ;
3. if it is parallel to at least a pair of states from $S_{k1} \cup S_{k2}$ and the appropriate entries of $\boldsymbol{u}_i$ have the orthogonal values, then the entry is “+.”

The ith entry of $\boldsymbol{u}^z_k$ defines whether the state s_i may be encoded with kth coding variable, and if yes (ith entry is not “+”), it shows what may be the value of that variable in the code. For example, for the automaton under consideration, boundary vectors for matrix U rows are displayed in the third column in Table 11-1.

Now we define some operations over 4-valued vectors that extend those over ternary vectors (keeping in view vectors of the same length). A 4-valued vector $\boldsymbol{b}$ is an inversion of a vector $\boldsymbol{a}$ if, whenever the ith entry of $\boldsymbol{a}$ is 1, 0, –, +, the ith entry of $\boldsymbol{b}$ is 0, 1, –, +. The 4-valued vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ (one of them can be ternary) are orthogonal if for at least an index i one vector has the ith entry equaled to $\sigma \in \{1, 0\}$, and the other to σ or “+.” A weight of a vector is defined as a sum of the weights of its entries, supposing that the weight of the entry equaled to 1, 0, –, + is 1, 1, 0, 2, respectively. A 4-valued vector $\boldsymbol{a}$ covers a vector $\boldsymbol{b}$ (4-valued or ternary) nonorthogonal to $\boldsymbol{a}$ if, whenever the ith entry of $\boldsymbol{b}$ is $\sigma \in \{1, 0-\}$, the ith entry of $\boldsymbol{a}$ is σ or ÷

Example 3. The vectors $-1{+}0{-}{+}0{-}1$ and $01{+}00{+}0{+}1$ are nonorthogonal and the first covers the second. Their weights are 8 and 12, respectively. The inversion of the first vector is $-0{+}1{-}{+}1{-}0$.

3.4 Compressing a set of encoding constraints toward the set of codes

When using the notion of a boundary vector we can simply find whether two rows of $\boldsymbol{\nu}_k \in \boldsymbol{V}$ and $\boldsymbol{u}_l \in \boldsymbol{U}$ (or both from $\boldsymbol{U}$) are compatible. That takes place if $\boldsymbol{\nu}_k^z$ is nonorthogonal to $\boldsymbol{u}_l$ or $'\boldsymbol{u}_l$.

When concatenating two rows $\boldsymbol{\nu}_k$ and $\boldsymbol{u}_l$ (constructing their implicant), we do minimal extensions of $\boldsymbol{\nu}_k$ to implicate $\boldsymbol{u}_l$ and of $\boldsymbol{\nu}_k^z$ to implicate $\boldsymbol{u}_l^z$ (or inversions of $\boldsymbol{u}_l$ and $\boldsymbol{u}_l^z$. In this way any ith entry of the result of concatenation is equal to that of $\boldsymbol{\nu}_k$ and $\boldsymbol{u}_l$ (or $'\boldsymbol{u}_l$), depending on whose weight is bigger. Any ith entry of the boundary vector of the result of concatenation is equal to any entry of $\boldsymbol{\nu}_k^z$ and $\boldsymbol{u}_l^z$ that has bigger weight or '+'if they are orthogonal.

$\boldsymbol{\nu}_k \in \boldsymbol{V}$ is an implicant for generalized constraint $\boldsymbol{U}_l \in \boldsymbol{U}$ if it is an implicant for one of the elementary constraints $\boldsymbol{u}_{li,j} \in \boldsymbol{u}_{li}^p \in \boldsymbol{U}_l$. An implicant of a subset U' of generalized constraints is maximal if it is incompatible with all those others (it cannot implicate any more constraints besides those from U').

Example 4. The boundary vectors for the vectors $\boldsymbol{u}_{2,2.1} = 10{-}{-}0{-}{-}{-}{-}$ and $\boldsymbol{u}_{7,2.1} = -1{-}{-}1{-}{-}{-}0$ (Example 1) are $\boldsymbol{u}_{2,2.1}^z = 10{+}00{+}0{-}{-}$ and $\boldsymbol{u}_{7,2.1}^z = -1{+}11{+}1{-}0$, respectively. $\boldsymbol{u}_{2,2.1}$ is compatible with the inversion of $\boldsymbol{u}_{7,2.1}$ and their implicant is $10{-}{-}0{-}{-}{-}1$ with boundary vector $10{+}00{+}0{-}1$. This implicant is not maximal, but $10{-}{-}0{-}{-}11$ is maximal.

4. SOLUTIONS OF THE PROBLEM OF RACE-FREE STATE ASSIGNMENT

The encoding matrix $\boldsymbol{V}$ is grown from an initial seed matrix $\boldsymbol{U}$ by concatenating its rows. The rows of encoding matrix $\boldsymbol{V}$ may be found among the maximal implicants of the rows of $\boldsymbol{U}$. The task is to find the set of maximal implicants of minimal cardinality that satisfies all generalized constraints from the matrix $\boldsymbol{U}$.

An exact algorithm to find a minimum solution of the problem of race-free state assignment is based on building a set C of all maximal implicants for constraint matrix $\boldsymbol{U}$ and then looking for a subset of $\boldsymbol{V} \subseteq C$ of minimal cardinality, such that for any generalized constraint $U_i \in \boldsymbol{U}$ there exists an implicant in $\boldsymbol{V}$ implicating it. The second part of the problem is reduced to a covering problem of Boolean matrix[7], as in the case with Quine table.

4.1 Maximal implicant retrieval

We use branch-and-bound algorithm to build all maximal implicants. Constraints are processed one by one in a predefined order choosing (at each step) one compatible with the current state of the implicant formed. If we exhaust such constraints we would start backtracking to a previous step to modify the solution and repeat searching.

The computational efforts can be reduced using a previously generated compatibility relation on the rows from $\boldsymbol{U}$.

Taking into account that any maximal implicant may satisfy only one of the simple constraints from each generalized constraint, they all can be regarded as pairwise incompatible. At each step of the algorithm it is enough to consider as candidates for concatenating only the rows compatible with all concatenated in the current implicant. Further search reduction can be received by sorting the constraints according to the degree of their incompatibility: the greater it is, the less is branching.

Example 5. For the automaton considered, there exist 15 maximal implicants shown in the second column in Table 11-2. They have been found searching

Table 11-2. Maximal implicants and the Quine table

	Implicants	Quine table
No	12345 6789	12345 67890 123456
1	1−1−−0−− −	−−−1−−−−−−−−−−−−−
2	1−0−−1−− −	−−−−−−−11−−−−−−−−−−
3	1000000−−	−11−1−−−−−−−−−−−−−−1−
4	1−0−−1−− −	−−1−1−−−−−−−−−−−−−−1−
5	10−0−−1−1	−−1−−−−−1−1−−11−−
6	10−0−−1−0	−−1−−−−−1−−1−11−−
7	10−−0−−11	−11−− −11−1−−1−−1
8	10−−0−−10	−11−− −11−−1−1−−1
9	10−00−0−1	−11−−−−−−−1−−−−−1−
10	11−−1−−00	1−−−−−−−−−1−−−−− −1
11	11−11−−−0	1−−−−−−−−−1−−−−−−−
12	10−11−110	−−−−−−−111−11−−−−
13	11−00−0−0	11−− −11−−−−−−− −1−
14	11−00−001	−1− − −11−1−−1−−1−
15	11−1−−0−0	1− − − − −1−−1− − − − −−

for maximal compatible sets of constraints using their pairwise compatibility relation shown in matrix form in the fourth column in Table 11-1 (the relation is symmetric, so it is enough to define only a part of the matrix below the principal diagonal, and the relation among constraints from the same generalized constraint can be not defined). At first the fifth and eleventh constraint ($\boldsymbol{u}_{3,1.1}$ and $\boldsymbol{u}_{5,3.1}$ are examined to generate maximal implicants since they are compatible with none of the other constraints.

Then the sixth constraint $\boldsymbol{u}_{3,1.2}$ is chosen (it is compatible with three constraints), and so on.

4.2 Covering problem statement

Once a set C of maximal implicants is found, the task is to extract from it a subset that satisfies all generalized constraints $U_k \subset U$. Every generalized constraint $U_k = \{u^p_{k1}, u^p_{k2}, \ldots, u^p_{kn}\}$ is satisfied as OR (quite enough for one of its simple constraint u^p_{ki} to be satisfied) and u^p_{ki} consist of one or two u_{ij} that are satisfied as AND (both u_{i1} and u_{i2} should be satisfied). These statements can be expressed logically (as is suggested in Ref. 5) by the formulas: $U_k = u^p_{k1} \vee u^p_{k2} \vee \ldots \vee u^p_{kn}$ and $u^p_{ki} = u_{i1} \cdot u_{i2}$. Substituting expressions u^p_{ki} into U_k and using the distributive law one can receive the conjunctive normal form $U_k = U^1_k \cdot U^2_k \cdot \ldots U^m_k$. Any U^i_k is a union of separate elementary constraints. For example, generalized constraint $U_2 = \{U^p_{2,1}, U^p_{2,2}\}$ (Example 2) is represented as $U_6 = U^1_6 \cdot U^2_6 = (u_{2,1.1} \vee u_{2,2.1}) \cdot (u_{2,1.2} \vee u_{2,2.1}) = (\{s_1, s_2; s_4\}$ or $\{s_1; s_2, s_5\})\cdot$ $(\{s_1; s_2, s_4\}$ or $\{s_1; s_2, s_5\})$.

Now the problem is stated in the form of Quine table Q. Its rows correspond to maximal implicants $C_i \in C$, and columns to conjunctive members U^i_k for all U_k. An entry (ij) of Q is 1 (marked) if C_i implicates jth conjunctive member. The task is to find the minimal number of rows covering all columns (every column should have 1 at least in one position corresponding to the rows chosen)[7].

Example 6. For the automaton example considered, there exist 16 conjunctive members. They are as follows:

$U_1 = u_1(u_{1,1})$;
$U_2 = u_2 \vee u_4(u_{2,1.1} \vee u_{2,2.1})$;
$U_3 = u_3 \vee u_4(u_{2,1.2} \vee u_{2,2.1})$;
$U_4 = u_5(u_{3,1.1})$;
$U_5 = u_6(u_{3,1.2})$;
$U_6 = u_7(u_{4,1})$;
$U_7 = u_8 \vee u_{10} \vee u_{11}(u_{5,1.1} \vee u_{5,2} \vee u_{5,3})$;
$U_8 = u_9 \vee u_{10} \vee u_{11}(u_{5,1.2} \vee u_{5,2} \vee u_{5,3})$;

$U_9 = u_{12}(u_{6,1});$
$U_{10} = u_{13} \vee u_{14}(u_{7,1} \vee u_{7,2});$
$U_{11} = u_{15} \vee u_{16}(u_{8,1} \vee u_{8,2});$
$U_{12} = u_{17}(u_{9,1});$
$U_{13} = u_{18} \vee u_{19}(u_{10,1} \vee u_{10,2});$
$U_{14} = u_{20}(u_{11,1.1});$
$U_{15} = u_{21}(u_{11,1.2});$
$U_{16} = u_{22}(u_{12,1}).$

These conjunctive members are assigned to the columns of the Quine table shown in the third column in Table 11-2. Permissible minimum number of rows that provide the Quine table cover is 5. One of the minimal covers presenting encoding for automaton considered consists of the rows 3, 6, 10, 13, and 1. So we find the following 5-component codes of partial states that provide the absence of critical races when the automaton operates:

$$\boldsymbol{V} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & - & - \\ 1 & 0 & - & 0 & - & - & 1 & - & 0 \\ 1 & 1 & - & 0 & 0 & - & 0 & 0 & 1 \\ 1 & 1 & - & - & 1 & - & - & 0 & 0 \\ 1 & - & 1 & - & - & 0 & - & - & - \end{bmatrix}$$

It should be noted that some entries of a matrix with values 0 or 1 can be substituted with value ‘don’t care’because of usage of maximal implicants. In our case the irredundant form of the coding matrix is

$$\boldsymbol{V} = \begin{bmatrix} 1 & - & 0 & - & - & 0 & - & - & - \\ 1 & 0 & - & 0 & - & - & 1 & - & 0 \\ 1 & 1 & - & 0 & 0 & - & 0 & 0 & 1 \\ 1 & 1 & - & - & 1 & - & - & 0 & 0 \\ 1 & - & 1 & - & - & 0 & - & - & - \end{bmatrix}$$

5. CONCLUSION

The suggested method solves the encoding problem exactly: it ensures that the number of variables to encode partial states is minimal. Unfortunately the problems considered are computationally hard. The growth of the computation time as the size of the problem is a practical limitation of the method to be used in computer-aided design systems. It can be used for solving encoding problems of moderate size obtaining after decomposing the whole big problem. Besides, the method can be useful for estimation of efficiency of heuristic encoding techniques[3].

REFERENCES

1. M. Adamski, M. Wegrzyn, Field programmable implementation of current state machine. In: *Proceedings of the Third International Conference on Computer-Aided Design of Discrete Devises (CAD DD'99)*, Vol. 1. Institute of Engineering Cybernetics of the of Belarus Academy of Sciences, Minsk, 4~~1~~2 (1999).
2. L.D. Cheremisinova, Implementation of parallel digital control algorithms by asynchronous automata. *Automatic Control and Computer Sciences*, **19** (2), 78~~8~~3 (1985).
3. L.D. Cheremisinova, Race-free state assignment of a parallel asynchronous automaton. *Upravlyajushchie Sistemy i Mashiny*, **2** 51~~5~~4 (1987) (in Russian).
4. L.D. Cheremisinova, PLC implementation of concurrent control algorithms. In: *Proceedings of the International Workshop "Discrete Optimization Methods in Scheduling and Computer-Aided Design"*. Republic of Belarus, Minsk, pp. 190~~1~~96 Sept. 5~~6~~ (2000).
5. Yu.V. Pottosin, State assignment of asynchronous parallel automata with codes of minimum length. In: *Proceedings of the International Workshop "Discrete Optimization Methods in Scheduling and Computer-Aided Design"*. Republic of Belarus, Minsk, pp. 202~~2~~06 Sept. 5~~6~~ (2000).
6. S.H. Unger, *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York (1969).
7. A.D. Zakrevskij, *Logical Synthesis of Cascade Networks*. Nauka, Moscow (1981) (in Russian) .
8. A.D. Zakrevskij, Parallel automaton. *Doklady AN BSSR*, **28** (8), 717~~7~~19 (1984) (in Russian).
9. A.D. Zakrevskij, *Parallel Algorithms for Logical Control*. Institute of Engineering Cybernetics of NAS of Belarus, Minsk (1999) (in Russian).

Chapter 12

DESIGN OF EMBEDDED CONTROL SYSTEMS USING HYBRID PETRI NETS

Thorsten Hummel and Wolfgang Fengler
Ilmenau Technical University, Department of Computer Architectures, P.O. Box 100565, 98684 Ilmenau, Germany; e-mail: wolfgang.fengler@tu-ilmenau.de

Abstract: The paper describes the challenges of modeling embedded hybrid control systems at a higher abstraction level. It discusses the problems of modeling and analyzing such systems and suggests the use of hybrid Petri nets and time interval Petri nets. Modeling an exemplary embedded control system with a special hybrid Petri net class using an object-oriented modeling and simulation tool and the extension of hybrid Petri nets with the concept of time intervals for analyzing time constraints shows the potential of this approach.

Key words: embedded control systems; hybrid Petri nets; time interval Petri nets.

1. INTRODUCTION

The design of complex embedded systems makes high demands on the design process because of the strong combination of hardware and software components and the observance of strong time constraints. These demands rise rapidly if the system includes components of different time and signal concepts. This means that there are systems including both event parts and continuous parts. Such systems are called *heterogeneous* or *hybrid* systems.

The behavior of such hybrid systems cannot be covered homogeneously by the well-known specification formalisms of the different hardware or software parts because of the special adaptation of these methods to their respective field of application and the different time and signal concepts the several components are described with. A continuous time model usually describes continuous components, whereas digital components are described by discrete events.

For describing both kinds of behavior in its interaction, there are different approaches to describe such systems. On the one hand, the different components can be described by their special formalisms. On the other hand, homogeneous description formalism can be used to model the complete system with its different time and signal concepts, and that is what we are in favor of.

Therefore, we have investigated modeling methods that can describe the behavior of such systems homogeneously at a high abstraction level independently from their physical or technical details. Apart from considering the heterogeneity, the modeling method must cope with the high complexity of the modeled system. This demand requires support for modularization and partitioning and hierarchical structuring capabilities. To meet the challenges of strong time constraints the tool used should have time analysis capabilities.

In the following sections, a graph-based formal modeling approach is presented. It is based on a special Petri net class, which has extended capabilities for the modeling of hybrid systems. To model the hybrid systems, we have used an object-oriented modeling and simulation tool based on this Petri net class. This tool can be used for modeling hybrid systems from an object-oriented point of view. It can be used for modeling and simulating components or subsystems and offers capabilities for hierarchical structuring.

By extending the used Petri net class with the concept of time intervals, an analysis method for time constraints could be implemented.

2. HYBRID PETRI NETS

The theory of Petri nets has its origin in C.A. Petri's dissertation "Communication with Automata"[7], submitted in 1962. Petri nets are used as describing formalism in a wide range of application fields. They offer formal graphical description possibilities for modeling systems consisting of concurrent processes. Petri nets extend the automata theory by aspects such as concurrency and synchronization.

A method to describe embedded hybrid systems homogeneously is the use of hybrid Petri nets[1]. They originate from continuous Petri nets introduced by David and Alla[2]. A basic difference between continuous and ordinary Petri nets is the interpretation of the token value. A token is not an individual anymore, but a real quantity of token fragments. The transition moves the token fragments from the pre-place to the post-place with a certain velocity of flow. The essence of hybrid Petri nets is the combination of continuous and discrete net elements in order to model hybrid systems.

In the past, there were applications of hybrid Petri nets described in many cases, but essentially they were concentrated on the fields of process control and automation. In the following section we demonstrate the possibilities of using hybrid Petri nets to model embedded hybrid systems. The used Petri net class of hybrid dynamic nets (HDN) and its object-oriented extension is described in Refs.3 and 4. This class is derived from the above-mentioned approach of David and Alla and defines the firing speed as function of the marking from the continuous net places.

Components or subsystems are modeled separately and abstracted into classes. Classes are templates, which describe the general properties of objects. They are grouped into class libraries. Classes can be used to create objects, which are called *instances* of these classes. If an object is created by a class, it inherits all attributes and operations defined in this class.

One of the important advantages of this concept is the ability to describe a larger system by decomposition into interacting objects. Because of the properties of objects, the modification of the system model could be achieved easily. The object-oriented concept unites the advantages of the modules and hierarchies and adds useful concepts such as reuse and encapsulation.

3. EXTENSION OF THE HYBRID PETRI NET APPROACH BY TIME INTERVAL CONCEPTS

The used hybrid Petri net approach includes discrete and continuous components. For analyzing time aspects the discrete transitions are extended by the concept of time intervals[6,7].

Time-related Petri nets were introduced for the integration of time aspects to classical Petri nets. There exist two basic types of time-related Petri nets. One type relates a firing time to the transition. The other type relates a time interval to the transition during which this transition is allowed to fire.

In our approach we use the latter. In a time interval Petri net, a transition with firing capability may fire only during the given interval and must have fired at the latest with the end of the interval, except that it loses its firing capability in the meantime. If the transition gets back its firing capability in the later time, the interval time starts again.

3.1 Analysis of a specification model by using time interval Petri nets

By using the implemented time interval Petri net approach, structural and behavioral properties of the modeled system can be analyzed. The analysis delivers information about properties such as reachability, liveness, and boundedness. To describe the state of a time interval Petri net, the time component has to be considered because of the change of the net behavior with different time intervals.

The analysis method consists of two steps. At first, for a given transition sequence the reachability graph has to be calculated without considering any time restrictions. In the next step a state class for the given transition sequence with consideration of the time intervals has to be built. Every class includes all states with the same marking components. The states of one class differ by its time components. A change in the marking component leads to a class change.

The method of class building is based on the investigation of firing capabilities for all transitions with the progressive time. The resulting inequation system of every class can be solved with methods of linear optimization considering additional conditions. Thereby for the given transition sequence, different time characteristics can be found:

- Worst case: maximum-minimum analysis,
- Observance of time constraints (deadline),
- Livelock.

The analyzing method is implemented with methods of linear optimization, the simplex algorithm, and the interior-point method. These methods can be used alternatively.

4. MODELING AN EMBEDDED CONTROL SYSTEM

The application example we have chosen to discover the possibilities of using hybrid Petri nets for modeling embedded hybrid control systems and analyzing time-related aspects of this system is an integrated multi-coordinate drive[8]. This is a complex mechatronic system including a so-called multicoordinate measuring system. Figure 12-1 shows this incremental, incident light measuring system consisting of three scanning units fixed in the stator and a cross-grid measure integrated into the stage. The two y-systems allow determining the angle of rotation φ. The current x, y_1, and y_2 position is determined by the cycle detection of its corresponding sine and cosine signals. The full cycle counter keeps track of completed periods of the incremental measuring system. This is a precondition for the following deep interpolation. The cycle counter of these signals is a function of the grid constant and the shift between the scanning grids and the measure. The cycle counter provides a discrete position, and in many cases this precision is sufficient for the motive control algorithm. To support a very precise position control with micrometer or nanometer resolution, it must be decided which possibility of increasing the measure precision is the most

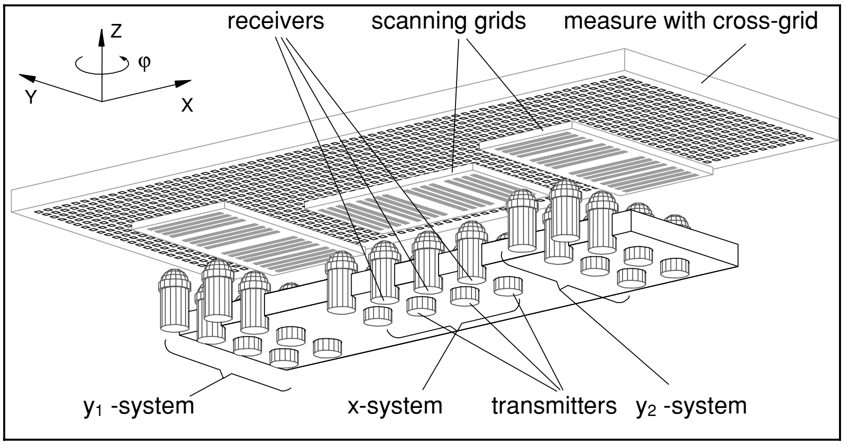


Figure 12-1. Multicoordinate measuring system.

cost-efficient. There is a limit to improving the optic and mechanical properties because of the minimum distances in the grid.

Alternatively, an interpolation within a signal period can be used, whereby the sampling rate of the A/D converter is increased, which would allow a more detailed evaluation of the continuous signals of the receiver. The problem to be solved in this application example leads to modeling and simulating the measure system together with the evaluation algorithm for the position detection.

The measuring system is hierarchically modeled using components (Fig. 12-2).

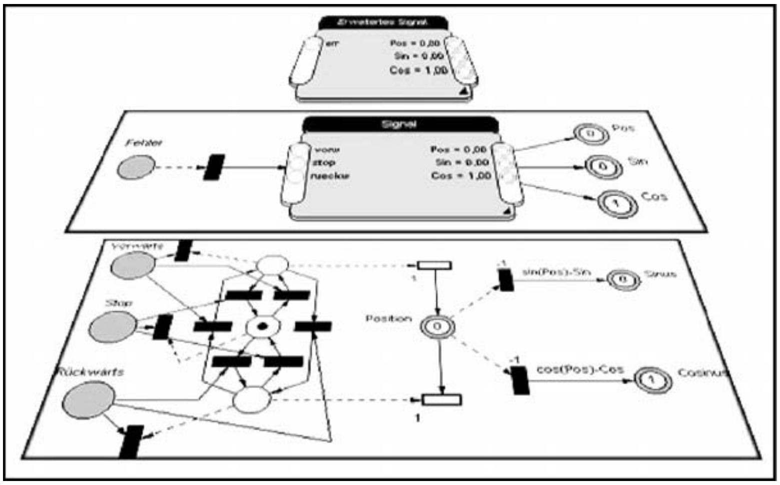


Figure 12-2. The principle of hierarchical modeling.

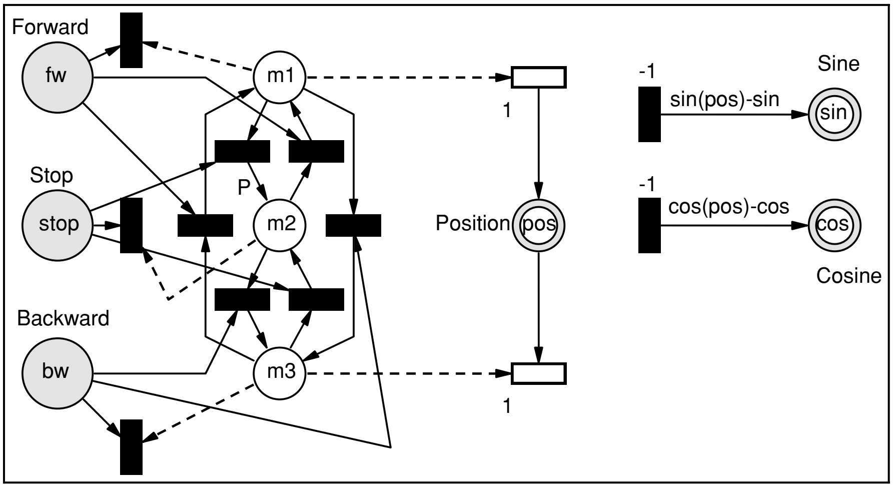


Figure 12-3. Component “Signal generation.”

Components with the same functionalities are abstracted into classes, put into a class library, and instantiated while modeling. The modeling of a multi-hierarchical system is possible as well.

4.1 System environment

The component “Signal generation” (Fig. 12-3) simulates the sensor data and provides the sine and cosine signals as well as a position value. For clearness reasons this net is saved as a component into a subnet and gets the input places “Forward,” “Stop,” and “Backward.” It provides a sine and a cosine signal and additionally a position signal as a comparative value for a later error control function.

To simulate a potential misbehavior of the measuring system, external disturbances are modeled in the subnet “Scrambler,” which is included in the component “Disturbance” of the complete system.

4.2 Measuring system components

The position detection of one axis is modeled with the component “Axis-measure” (Fig. 12-4).

At first, the input signals “sine” and “cosine” are normalized in the subnets “minmax_s” and “minmax_c.” These subnets are identical in their functions and were instantiated during the modeling process from the same class “minmax” (Fig. 12-5). To find out the exact position of the carrier, the cycle number has to be determined in “meas_1.” To determine this correctly, the measuring system

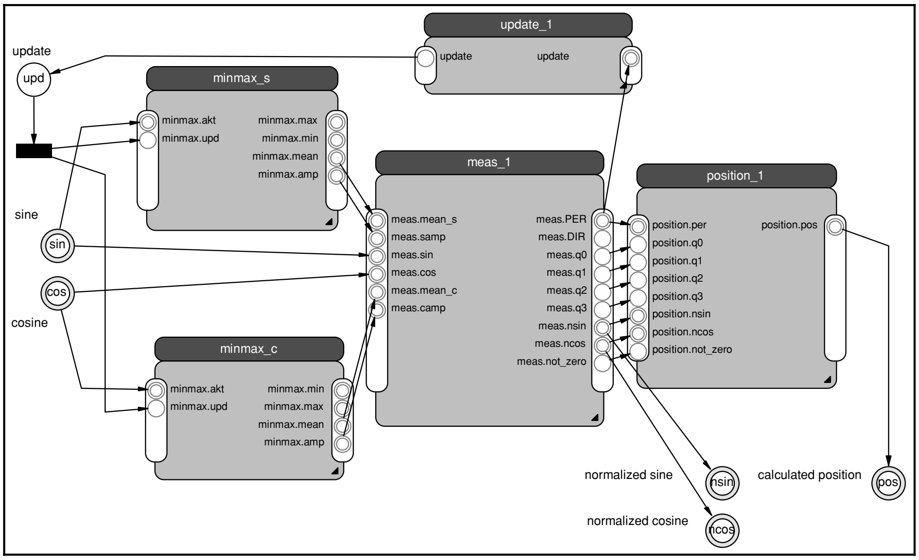


Figure 12-4. Subnet "Axismeasure."

has to detect the moving direction of the carrier and with it the increasing or decreasing of the cycle number. The original measuring system used a look-up table, but this was very hard to model with Petri nets. Therefore, we changed this into logic rules and used this to model the subnet "position _1."

4.3 Model of the entire system

In Fig. 12-6, the model of the entire system is shown. Besides the measuring system, it includes the components for signal generation and external

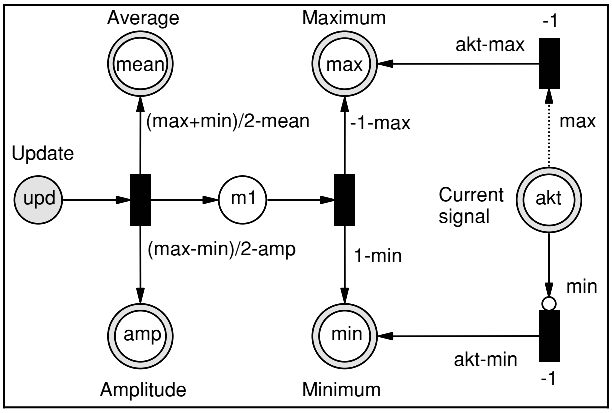


Figure 12-5. Subnet "Minmax."

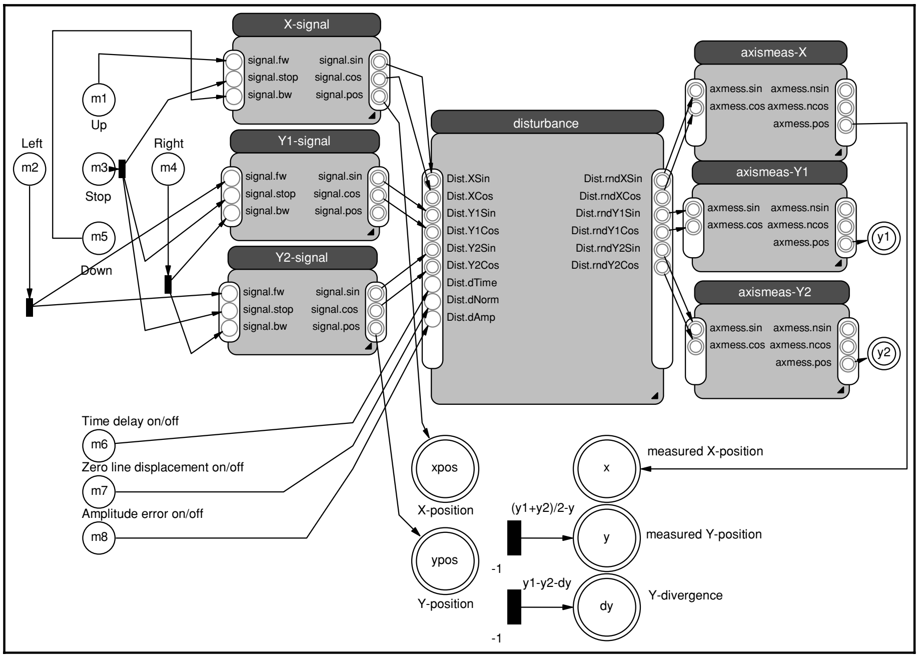


Figure 12-6. Model of the entire system.

disturbance simulation. The components for signal generation "$x/y_1/y_2$-direction"are instances of the class "Signal"and model the signals of an ideal environment.

The component "Disturbance"includes the simulation of various kinds of signal disturbances (displacement of the zero line, amplitude errors, time delay, etc.). The signal disturbances can be turned on and off at any time during the simulation.

The objects "Axismeasure $\_x/y_1/y_2$"are based on the class "Axismeasure" and include the evaluation algorithm for the three directions. The motion of any desired direction can be controlled by feeding marks into the places m1 to m8.

The x-position, the average y-position, and the divergence of the y-position arose as result of the net calculation.

4.4 System simulation

The tool "Visual Object Net++" [5] allows not only the modeling but also the simulation of systems described with hybrid dynamic nets. During the simulation the firing of the transitions and the transport of the tokens are animated. The changes of the place values can be visualized by signal diagrams (Fig. 12-7).

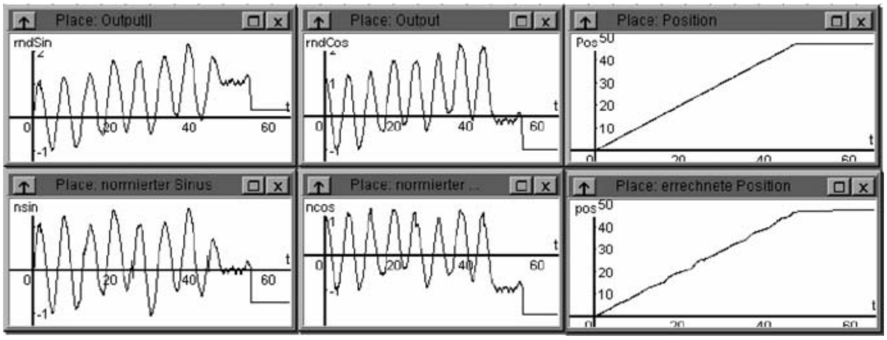


Figure 12-7. System behavior with different disturbances.

For example, the middle top diagram in Fig. 12-7 shows an extreme example of a simulation with disturbances. It shows a clear exceeding of the zero line of the cosine signal. Nevertheless, the normal values are correctly calculated and the position of the machine is correctly displayed.

4.5 System analysis

The modeling and simulation tool ‘VisualObjectNet++’ was extended by an analysis component for time constraints. In this application example, a time interval Petri net for the component ‘Axismeasure’ was made (Fig. 12-8). For a defined transition sequence and given intervals, a maximum-minimum analysis was made to determine the worst-case behavior of this system component. The analysis results in the fulfillment of the given restrictions.

5. CONCLUSION

Our investigation has shown the advantages of using hybrid Petri nets for homogeneous modeling of an embedded hybrid system. The object-oriented approach of the hybrid Petri net class used makes possible a clear modeling of complex hybrid systems.

The analysis of time-related properties of complex embedded systems offers the chance to check, in early stages of the design flow, if the modeled system matches to given time constraints.

Things that have to be done in future are the extension and completion of the system model and the integration of the modeling process in a complete design flow. Here we focus our future work on connecting our approach to

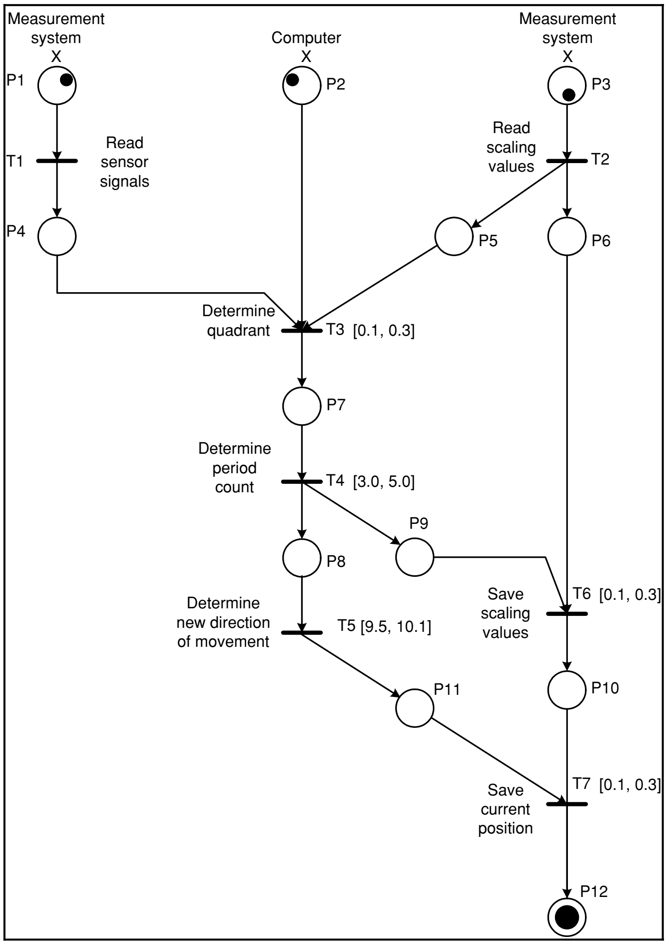


Figure 12-8. Interval Petri net of the analyzed system component.

other approaches related to hardware/software partitioning and SoC design on system level.

ACKNOWLEDGMENT

This research work was supported by the DFG (Deutsche Forschungsgemeinschaft, German Research Association) as part of the investigation project Design and Design Methodology of Embedded Systems”with the subject Design of Embedded Parallel Control Systems for Integrated Multi-Axial Motive Systems’under grant FE373/13-1/2.

REFERENCES

1. H. Alla, R. David, J. Le Bail, Hybrid Petri nets. In: *Proceedings of the European Control Conference*, Grenoble (1991).
2. H. Alla, R. David, Continuous Petri nets. In: *Proceedings of the 8th European Workshop on Application and Theory of Petri nets*, Saragossa (1987).
3. R. Drath, Modeling hybrid systems based on modified Petri nets. PhD Thesis, TU Ilmenau, (1999) (in German).
4. R. Drath, Hybrid object nets: An object-oriented concept for modeling complex hybrid systems. In: *Hybrid Dynamical Systems. Third International Conference on Automation of Mixed Processes*, *ADPM'98*, Reims (1998).
5. L. Popova-Zeugmann, Time Petri nets. PhD Thesis, Humboldt-Universität zu Berlin (1989) (in German).
6. L. Popova-Zeugmann, *On Parametrical Sequences in Time Petri Nets*. Humboldt-University Berlin, Institute of Informatics.
7. C.A. Petri, *Communication with Automata*. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn (1962) (in German).
8. E. Saffert, C. Schäffel, E. Kallenbach, Control of an integrated multi-coordinate drive. In: *Mechatronics'96*, Vol. 1, Guimaraes, Portugal, Proceedings 151–156 September 18–20, (1996).

Section IV

Implementation of Discrete-Event Systems in Programmable Logic

Chapter 13

STRUCTURING MECHANISMS IN PETRI NET MODELS

From specification to FPGA-based implementations

Luís Gomes[1], João Paulo Barros[1,2], and Anikó Costa[1]
[1]Universidade Nova de Lisboa / UNINOVA, Monte de Caparica, Portugal
[2]Instituto Politécnico de Beja, Escola Superior de Tecnologia e Gestão, Beja, Portugal;
e-mail: lugo@uninova.pt, jpb@uninova.pt, akc@uninova.pt

Abstract: This chapter addresses the use of modular model structuring mechanisms for the design of embedded systems, using reactive Petri nets. Relevant characteristics of reactive Petri nets are briefly presented. One graphical hierarchical structuring mechanism named *horizontal* decomposition is presented. This mechanism relies on the usage of macronodes, which have subnets associated with them and can be seen as a generalization of widely known mechanisms available in several Petri nets tools. Three types of macronodes are used: macroplace, macrotransition, and macroblock. The model execution is accomplished through the execution of the model's flat representation. Additional folding mechanisms are proposed through the introduction of a vector notation associated with nodes and external signals. A running example of a controller for a 3-cell first-in-first-out system is used illustrating the several modular construction mechanisms. High-level and low-level Petri net models are used and compared for this purpose. A modular composition operation is presented and its use in the controller's design is exemplified. Finally, an overview of distinct field programmable gate array (FPGA)-based implementation strategies for the referred controller is discussed.

Key words: Petri nets; structuring mechanisms; modular design; programmable logic devices; field programmable gate arrays.

1. INTRODUCTION

Petri nets are a well-known formal specification language offering numerous and important advantages for the modeling of concurrent and distributed systems[10]. Among them we emphasize three: *locality*, *concurrency*, and *graphical representation*. Locality means the model's evolution is based on transition firing, and the firing of a specific transition is evaluated exclusively on the basis of its attributes and its input and output arcs and places. Concurrency is implicit as transitions can fire independently (if they have disjointed associated conditions).

The graphical representation often allows a more readable model provided some care is taken to decompose large models in smaller parts. In fact, although the importance of a modular system structuring has been recognized since a long time, Petri nets have no intrinsic support for modular structuring. This has motivated several proposals for the modular structuring of Petri net models[17,4,16,18,8,6,12,13]. These proposals allow different levels of abstraction enabling an incremental model construction.

From our point of view, the concepts proposed in Huber et al.[17], namely substitution transitions and substitution places, are of utmost importance. The substitution transition concept is probably the most used hierarchical structuring mechanism, as it is also used by the two main coloured Petri nets tools: CPNTools and Design/CPN. The substitution place mechanism plays a dual role related with substitution transitions. In both cases, the model execution is accomplished through the flat model obtained after the fusion of nodes in one net level with nodes in the other net level. It is important to stress that, in Huber et al.[17], one is not allowed to directly connect a substitution place to a substitution transition. Moreover this was identified but was considered not to be a serious restriction. We disagree with that assumption and we will consider a more flexible hierarchical structuring mechanism, supporting modular specifications. Some other model-structuring mechanisms, important for embedded systems specification, are not used in this chapter. Among them, one can mention the concept of depth, extensively used in statecharts[15], and also integrated in some Petri net classes.

The chapter extends a previous paper on hierarchical structuring mechanisms for Petri nets, for use in the programmable logic devices (PLD)-based systems[12], with a proposal for modular addition of Petri net models and a discussion about several strategies for their implementation in PLDs, namely in field programmable gate arrays (FPGAs). The chapter starts by a brief presentation of a Petri net class supporting hierarchical structuring of Petri net models. Then we present a running example, which is used to show several distinct and alternative ways to compose Petri net models. Next, horizontal decomposition is presented and the system structure is shown to be specifiable in a precise way by the use of an operation for the modular addition of Petri net models named *net addition*. The chapter ends with a discussion about several possible strategies to implement net models in FPGAs.

2. REACTIVE PETRI NET

In the present chapter, we will use as reference a nonautonomous high-level Petri net class, named *reactive petri nets*[11] (RPN). This class of Petri nets is briefly and informally presented in this section.

Its characteristics are divided into two groups: the autonomous part, dealing with the intrinsic graph characteristics; and the nonautonomous part, modeling of: inputs, outputs, time, priorities, conflict resolutions, and other implementation-specific issues. The name *reactive* for the Petri net class comes from the proposed semantics for transition firing.

Because of their intrinsic characteristics, coloured Petri nets[18] were chosen as the reference model for the autonomous part of the model. They have two main advantages: the capability to create compact models; and the possibility to specify data processing, in addition to control flow. The former allows a useful treatment of structural and marking symmetries found in many real-world applications, while the latter complements the control-dominated nature of low-level Petri nets. This is very desirable for embedded systems modeling. The data processing specification is supported by transition bindings, guards, and arc inscriptions.

Synchronized Petri nets and interpreted Petri nets[19,9], as well as statecharts[15], are the reference paradigms for the nonautonomous part in RPNs. However, in this chapter, only the extensions associated with input/output modeling are important.

Modeling of input events is accomplished through their usual association with transitions. In this sense, input is specified by event conditions associated with transitions. The transition firing rule was changed so as to consider the new input dependency. As such, for a transition to fire it must fulfill two conditions: it must be *enabled* (by the existence of a specific binding of tokens presented in the input places), and it must be *ready* (the external input evaluation must be true). Every enabled and ready transition will be fired. This means that the maximal step is always used. This is the approach followed by the already referred synchronized Petri net and interpreted Petri net classes.

Output actions can be associated with either transition firings or place markings. In the former case, transition firing generates output events (as in a Mealy automaton). In the latter case, output actions are generated according to the place markings (as in a Moore automaton). Outputs can be made dependent on specific marking or binding attributes, as convenient. As an example, an output associated with a place can be updated only if a specific attribute of the coloured token stored in the place satisfies a specific condition.

3. AN EXAMPLE

In this section we present a running example that allows us to illustrate the structuring and composition mechanisms that we will introduce. It is a 3-cell first-in-first-out (FIFO) system with four associated conveyor belts. This is a simplified version of an example presented elsewhere[14].

Each conveyor (except the last one) has sensors to detect objects on its inputs and outputs. These sensors have associated variables *IN[1..4]* and *OUT[1..3]*. Each conveyor also has movement control, through the variables *MOVE[1..3]*. Figure 13-1 presents the referred layout.

Depending on the designer preferences, several modeling styles can be used, e.g., starting with a high-level Petri net model, as in Fig. 13-2, or using low-level nets, as in Fig. 13-3. In Fig. 13-2, as a simplified notation, the ith element of the input/output signal vector x, $x[i]$, is referred to as xi. As a matter of fact, Fig. 13-3 can be seen as a (slightly modified) unfolding of the model in Fig. 13-2. In Fig. 13-3, we can easily identify three model parts associated with different token colors in Fig. 13-2. Places and transitions in Fig. 13-3 exhibit the names used in Fig. 13-2 and additionally receive a vector index. This will be presented later on.

The colored model in Fig. 13-2 easily accommodates the modeling associated with the system's expansion. For example, if one wants to model a 25-cell FIFO system, the changes are really easy to follow, as far as only initial marking at *pa* and *p6*, and guards at *t2* and *t3*, is changed accordingly.

On the other hand, the equivalent low-level Petri net model will expand through several pages, although as a result of the duplication of a specific submodel. Yet, for an implementation based on "elementary platforms" (those without sophisticated computational resources), it is probably preferable to start from the low-level model, as it can be used directly as an implementation specification. This is the case for hardware implementations, namely the ones supported by FPGAs (or other reconfigurable devices), where each node can be

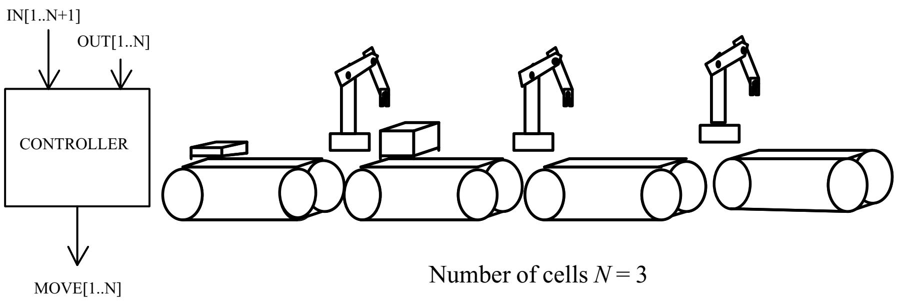


Figure 13-1. N-cell FIFO system model (with $N = 3$).

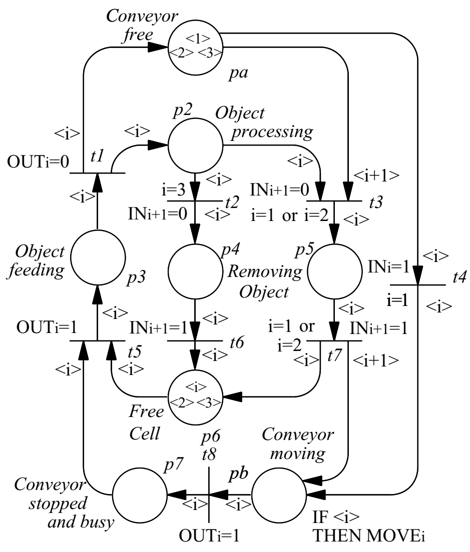


Figure 13-2. High-level Petri net model of the FIFO system.

directly translated to implementation: places as flip-flops or registers, transitions as combinatorial logic functions.

In this sense, it is important to allow compactness of the model while keeping the low-level modeling attitude. This is one major goal of the proposals in this chapter. To this end, let us start by identifying, in Fig. 13-4, the submodel that has to be replicated for each cell that we want to add to the FIFO system. This attitude uses the common structuring mechanism usually referred as macronode, or horizontal decomposition, and widely used within low-level nets (i.e., in automation applications). It is also used by colored Petri nets and associated tools as part of the hierarchical colored Petri net class[18].

We name the nodes on the border of this submodel *interface nodes*. It has to be noted that the model in Fig. 13-3 can be obtained by a triple replication of the submodel presented in Fig. 13-4 and the addition of one transition and one place. The added transition represents the initial feeding of the system with an object in the first conveyor; the added place represents the objects that exit the system (as one last conveyor).

It has to be noted that places *ppa* and *ppb* of *Cell[i]* are fused with places *pa* and *pb* of *Cell[i + 1]*, respectively. In this sense, *ppa* and *ppb* can be seen as dummy places used to allow submodels to be glued together and to offer the support for the addition of an arc from place *pa* of *Cell[i]* to transition *t*3 of *Cell[i − 1]*. Therefore, places *ppa*/*ppb* are graphical conveniences to represent the arcs to or from transition *t3* or *t7*, by this sense, we can represent the

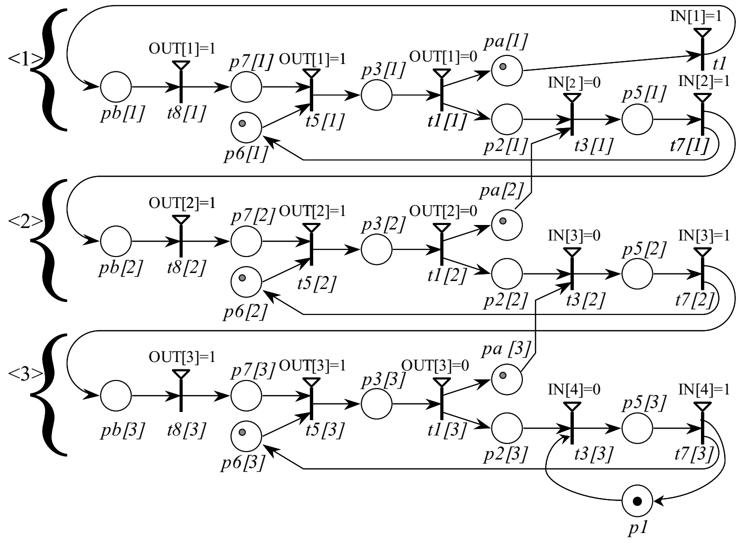
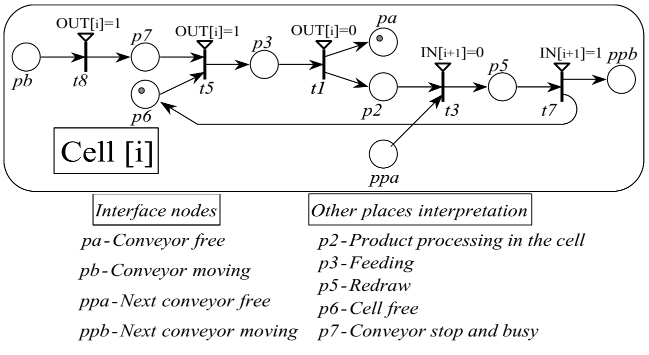


Figure 13-3. Low-level net model for the FIFO system.

submodel, in an encapsulated form, by one node (named macroblock). This allows the higher-level representation shown in Fig. 13-5.

As a final folding step, we may take advantage of the regularity of the model and also of the vectorial notation we have been using for the representation of external signals, places, and transitions. This makes possible the compact model in Fig. 13-6. As in the colored model, it is easy to accommodate the modeling of a 25-cell FIFO system: we only need to replace “3” by “25” in three arc inscriptions and change the vector dimension.

Typically, the design of Petri net models starts with the construction of a graphical specification for the system model, but afterward we need to represent it formally (to feed our verification and implementation tools). The



Interface nodes	Other places interpretation
pa - Conveyor free	*p2 - Product processing in the cell*
pb - Conveyor moving	*p3 - Feeding*
ppa - Next conveyor free	*p5 - Redraw*
ppb - Next conveyor moving	*p6 - Cell free*
	p7 - Conveyor stop and busy

Figure 13-4. Submodel of a cell.

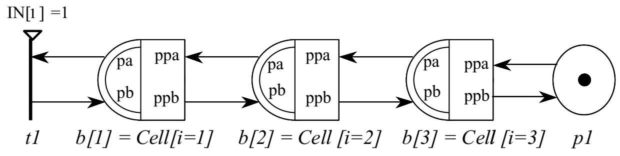


Figure 13-5. Model of the system using macroblocks.

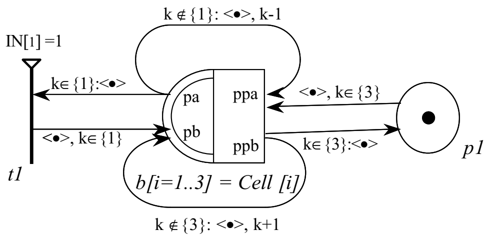


Figure 13-6. Folded model of the system based on a vector of macroblocks.

formalization of the net composition, including the use of macronodes for hierarchical structuring, can be supported by a small set of operations (defined elsewhere[1,13]). The basic operation is named *net addition*.

Net addition is an intuitive operation that works in two steps. In the first step, two nets are considered as a single net (a disjoint union). In the second step, several pairs of nodes (either places or transitions) are fused. Figure 13-7 exemplifies net addition. The addition of the two upper nets, *Start* and *Cell*, results in net *StartCell*. Roughly speaking, the addition was accomplished through the

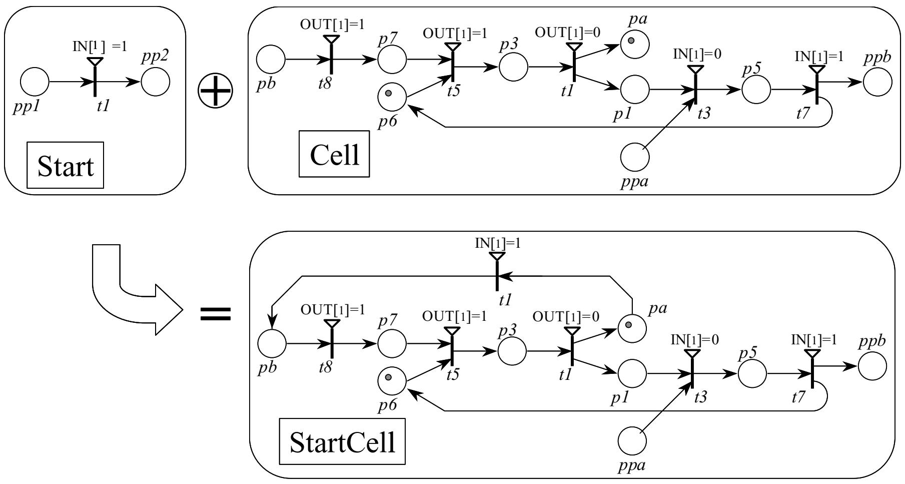


Figure 13-7. Addition of nets.

fusion of two pairs of places: *pp1* from the *Start* net with *pa* from the *Cell* net and *pp2* from the *Start* net with *pb* from the *Cell* net.

The operation is amenable to be algebraically represented as

$$\text{StartCell} = \text{Cell} \oplus \text{Start}\,(\text{pa/ppl} \rightarrow \text{pa}, \text{pb/pp2} \rightarrow \text{pb}) \tag{1}$$

where the places preceding the arrows are the interface nodes. The nodes after the arrows are the resulting merged nodes. For example, the notation *pa/pp1→ pa* means that nodes *pa* and *pp1* are merged (fused) together giving origin to a new node named *pa*.

Also, the whole model presented in Fig. 13-3 can be straightforwardly produced through the following expression, where *Place* represents a subnet with no transitions and a single place *p1* containing one token[*]:

$$\begin{aligned}\text{System} = {} & \text{Place} \oplus \text{Cell}[3] \oplus C\text{ell}[2] \oplus \text{Cell}[1] \oplus \text{Start}\\ & (\text{Cell}[1].\text{pa/pp} \rightarrow \text{pa}[1], \text{Cell}[1].\text{pb/pp2} \rightarrow \text{pb}[1],\\ & \text{Cell}[1].\text{ppb/Cell}[2].\text{pb} \rightarrow \text{pb}[2], \text{Cell}[1].\text{ppa/Cell}[2].\text{pa} \rightarrow \text{pa}[2],\\ & \text{Cell}[2].\text{ppb/Cell}[3].\text{pb} \rightarrow \text{pb}[3], \text{Cell}[2].\text{ppa/Cell}[3].\text{pa} \rightarrow \text{pa}[3],\\ & \text{Cell}[3].\text{ppb/Cell}[3].\text{ppa/p1} \rightarrow \text{p1})\end{aligned} \tag{2}$$

First, the expression clearly shows the nets and the net instances that are added together. Then the expression shows the node fusions used to "glue" the nets together. We name this list of node fusions expressions a *net collapse* and each of the elements in the list a *named fusion set*. These two concepts, as well as net addition, are informally defined in section 5.

4. HORIZONTAL DECOMPOSITION

The horizontal decomposition mechanism is defined in the "common" way used in top-down and bottom-up approaches, supporting refinements and abstractions, and is based on the concept of module. The module is modeled in a separated net, stored in a page; every page may be used several times in the same design as a *net instance*. The pages with references to the modules are referred to as *superpages* (upper-level pages), while the pages containing the module model are referred to as *subpages* (lower-level pages). The nodes of the net model related with hierarchical structuring are named by macronodes. Three types of macronodes are used: macroplace, macrotransition (also used by hierarchical colored Petri nets[18]), and macroblock. Every macronode has an

[*]Note that, for simplification purposes, in Fig. 13-3, the nodes named *Cell[i].n* appear as $n[i]$.

associated subpage that can be referred to as a *macronet*. Distinctive graphical notations are used for the representation of macronodes: macroplaces are represented by a double circle (or ellipse), macrotransitions by a rectangle, and macroblocks are represented in a half-macroplace and half-macrotransition manner. They correspond to modules in a more general way. In this sense, the mechanism can be seen as a generalization of the macrotransition and macroplace concepts available in several tools. The hybrid nature of the macroblock node enables the existence of arcs connecting similar types of nodes (as shown in previous figures), although the bipartite intrinsic characteristic of Petri nets is not violated, as this representation is not an executable specification (at this level). In fact, the model's execution is accomplished through the flat model, which was the motivation for the name of this decomposition type. Finally, we note that this type of hierarchical decomposition is based solely in the autonomous characteristics of the model.

5. NET ADDITION

In this section we informally present net addition and the two main associated concepts: *net collapse* and *named fusion sets*. The respective formal definitions can be found elsewhere[3], together with a more comprehensive presentation. An implementation of these concepts is also presented elsewhere[2]. It is proposed in accordance with the Petri Net Markup Language (PNML)[5], which is currently the major input for a transfer format for the International Standard ISO/IEC 15909 on high-level Petri nets.

For simplification purposes, we base our presentation in terms of a low-level marked Petri net. This implies that all the nonautonomous extensions as well as the high-level features are not mentioned. In fact, for net inscriptions, only the net marking is taken into account: the respective place markings are added when places are fused. This assumes the existence of a net addition operation defined for net markings. All other net inscriptions can be handled in a similar manner as long as the respective addition operation is defined on them.

We now present informal definitions for a few concepts implicit in a net addition operation.

Given a net with a set of places P and a set of transitions T, we say that a *fusion set* is a subset of P or a subset of T. A fusion set with n nodes $x_1, x_2 \ldots, x_n$ is denoted as $x_1/x_2/\ldots/x_n$.

As the objective of fusion sets is the specification of a new fused node, we also define *named fusion sets*. A named fusion set is simply a fusion set ($x_1/x_2/\ldots/x_n$) plus the name for the node resulting (*rn*) from the fusion of all the nodes in the fusion set. It is denoted as $x_1/x_2/\ldots/x_n \rightarrow rn$. This notation was already used in the (1) and (2). We also say that *Nodes* $(x_1/x_2/\ldots/x_n \rightarrow rn) = \{x_1/x_2, \ldots, x_n\}$, and *Result* $(x_1/x_2/\ldots/x_n \rightarrow rn) = rn$.

Fusion sets and named fusion sets constitute the basis for the *net collapse* definition: a net collapse is simply a list of named fusion sets. For example, (1) uses a net collapse containing two named fusion sets.

The application of a net collapse *CO* to a net *N* is named a *net collapse operation*. Informally, each named fusion set *nfs*, in a net collapse *CO*, fuses all the nodes in *Nodes(nfs)* into the respective single node *Result(nfs)*. A *net addition* between two or more nets is defined as a net disjoint union of all the nets, followed by a *net collapse operation* on the resulting net.

Finally, the interested reader should refer to Refs. 2 and 3, where an additional reverse operation named *net subtraction* is also defined. This allows the undoing of a previous addition operation in a structured way.

6. VECTORS EVERYWHERE

The example presented already illustrated the use of net instances, which are specified by the net name followed by an index number between square brackets (e.g. *Cell[2]*). This is called the *net instance name*. The rationale for this notation comes from viewing each net as a template from which any number of instances can be created. Two instances are made different by the respective instance numbers. All the node identifiers and all the net inscriptions of a given net instance become prefixed by the net instance name (e.g. *Cell[2].p6*). This guarantees that all the elements in two net instances of a given net are disjoint.

The use of net instances is made particularly useful if we allow the definition of net vectors and iterators across the net vector elements. A *net vector* is simply a list of net instances: *Cell[1..3] = (Cell[1], Cell[2], Cell[3])*.

A *vector iterator* is defined as an integer variable that can be specified together with a vector declaration. This is used in the specification of net collapses. In this way we can specify collapse vectors, which allow the gluing of an arbitrary number of net instances. As an example, (3) generalizes (2) to any number of cells ((2) corresponds to *NCells* = 2):

$$
\begin{aligned}
\text{System} = {} & \text{Place} \oplus \text{Cell}[1..\text{NCells}] \oplus \text{Start} \\
& (\text{Cell}[1].\text{pa}/\text{pp1} \rightarrow \text{pa}[1], \text{Cell}[1].\text{pb}/\text{pp2} \rightarrow \text{pb}[1], \\
& (\text{Cell}[\text{i}].\text{ppb}/\text{Cell}[\text{i}+1].\text{pb} \rightarrow \text{pb}[\text{i}+1], \\
& \text{Cell}[\text{i}].\text{ppa}/\text{Cell}[\text{i}+1].\text{pa} \rightarrow \text{pa}[\text{i}+1]) \\
& [\text{i} : 1..\text{NCells} - 1], \\
& \text{Cell}[3].\text{ppb}/\ \text{Cell}[3].\text{ppa}/\text{p1} \rightarrow \text{p1})
\end{aligned}
\qquad (3)
$$

Especially important is the fact that (3) allows the specification of a similar system with any number of cells. Note the use of an iterator variable i in the specification of the addition collapse.

Vectors are also extremely useful for the input events. If we define an input event vector with *NIE* elements *ie[NIE]*, we can then use a vector element *ie[i]* as an event. If the index value i is specified as a (colored) transition variable, it can then be used to bind i. This is another way for the execution environment to interfere with the net execution. In this sense, transition firing depends not only on the marking of the connected places (as common on a Petri net), constrained by the guard expressions, but also on the occurrence of the associated events[12].

7. DISCUSSION ON IMPLEMENTATION ISSUES

Although the complexity of the running example is relatively small, it is amenable to exercise different possible implementation solutions. Designers can follow several implementation strategies – from direct implementations to indirect ones, including some "hybrid implementations" and from colored models to low-level nets. The appeal of the running example presented is that it allows simple comparisons between different ways to implement Petri nets models in hardware. Also, some of the tested solutions can use a microprocessor IP core to be embedded into FPGA designs. This allows us to exercise hardware-software codesign techniques within Petri net implementations.

As a common framework we had considered hardware implementations based on programmable logic devices; for most situations, complex programmable logic devices (CPLDs) can be adequate (or even simple devices like PALs for some solutions), but for some implementations, FPGA usage is mandatory (i.e., when we intend to use IP cores).

As another common characteristic for the tested implementations, VHDL coding was used for FPGA and CPLD implementations. This choice was due to the expressiveness capabilities of VHDL and also to its support for portability among different environments. Having said this, we have to add that all implementations were tested using Xilinx devices, some of them with CPLDs XC9536 and XC95108, but most of them with Spartan2 FPGA X2S200.

Several of the strategies referred to are very straightforward to implement. In the following paragraphs we summarize the main solutions we have tested or analyzed:

- The most intuitive implementation technique is to face direct implementation of every node of the low-level model; taking Fig. 13-3 as reference, each place can be implemented by a flip-flop (because it is a safe model), and each transition can be implemented as a combinatorial logic function (also dependent of external input signals).

- One alternative implementation solution, applicable to small complexity models, is the behaviorally equivalent state space, normally used for analysis and propriety verification. This alternative system's representation can be generated using a computer tool and can be implemented as an ordinary state machine. It is to be noted that even for relatively simple examples, such as the one presented, the associated state space is too large to be managed without the support of adequate tools (this was the main reason why it was not viable to exercise this implementation technique for the example). Note that for hardware direct implementations, any kind of state encoding technique can be used, from global encoding schemes to one-hot encoding; yet, the state machine can also be implemented in software, which can, in turn, be embedded into the FPGA through a microcontroller IP core (e.g., the *picoblaze* core[7] was successfully used).
- Alternatively, we may decompose the model into a set of interacting sub-models that will be executed concurrently; one solution using this attitude relies on the module concept associated with the macroblock previously identified in Fig. 13-4 (readily supported by the implementation language, VHDL); also, the usage of place invariant analysis led us to the conclusion that the five place invariants cover all the places of the model, and each invariant can be seen as a state machine (as far as only one place of the invariant is marked at a time). In this sense, the system's implementation could be based on the partitioning of the model into five concurrent state machines, which were trivially implemented, coded in either hardware (using VHDL) or software (using an assembler, with the support of the *picoblaze* IP).
- Finally, we may also decide for the direct implementation of the colored Petri net model, taking advantage of the existence of no conflicts in the model; to that end, places can be implemented using registers, and each transition can be implemented through a microcontroller IP core, which supports the data transformation characteristics associated with transitions.

8. CONCLUSIONS

In this paper, Petri net-based digital system design was addressed. Modular design was supported by the concept of the net addition operation. A hierarchical structuring mechanism, named horizontal decomposition, was presented on the basis of the concept of module, which can be represented by special kind of nodes, named macronodes, and was complemented by the vectorial representation of nodes and signals. Their usage was successfully validated through an example of a controller for a low-to-medium complexity system, which was implemented on the basis of programmable logic devices (normally

FPGAs, although CPLD- or even PAL-based implementations were carried on).

REFERENCES

1. João Paulo Barros, Luís Gomes, Modifying Petri net models by means of crosscutting operations. In: *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'2003)*; Guimarães, Portugal (18–20 June 2003).
2. João Paulo Barros, Luís Gomes, Operational PNML: Towards a PNML support for model construction and modification. In: *Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets; Satellite event at the International Conference on Application and Theory of Petri Nets 2004*, Bologna, Italy (June 26, 2004).
3. João Paulo Barros, Luís Gomes, Net model composition and modification by net operations: A pragmatic approach. In: *Proceedings of the 2nd IEEE International Conference on Industrial Informatics (INDIN'04)*, Berlin, Germany (24–26 June, 2004).
4. Luca Bernardinello, Fiorella De Cindio; A survey of basic net models and modular net classes. In: *Advances in Petri Nets 1992*; Lecture Notes in Computer Science, G. Rozenberg (ed.). Springer-Verlag (1992).
5. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, Michael Weber, The Petri net markup language: Concepts, technology, and tools. In: W. van der Aalst, E. Best (ed.), *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, LNCS, Vol. 2679, p. 483–505, Eindhoven, Holland, Springer-Verlag. (June 2003).
6. Peter Buchholz, Hierarchical high level Petri nets for complex system analysis. In: R. Valette (ed.) *Application and Theory of Petri Nets 1994, Proceedings of 15th International Conference*, Zaragoza, Spain, Lecture Notes in Computer Science Vol. 815, pp. 119–138, (1994) Springer-Verlag.
7. Ken Chapman; "Picoblaze"; www.xilinx.com
8. Søren Christensen, N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. *Application and Theory of Petri Nets 1994, Proceedings of 15th International Conference*, Zaragoza, Spain, Lecture Notes in Computer Science Vol. 815, (1994), pp. 159–178.
9. R. David, H. Alla, *Petri Nets & Grafcet; Tools for Modelling Discrete Event Systems*. Prentice Hall International (UK) Ltd; ISBN 0-13-327537-X (1992).
10. Claude Girault, Rüdiger Valk, *Petri Nets for Systems Engineering—A Guide to Modelling, Verification, and Applications*, Springer-Verlag, ISBN 3-540-41217-4 (2003).
11. Luis Gomes, Redes de Petri Reactivas e Hierárquicas—Integração de Formalismos no Projecto de Sistemas Reactivos de Tempo-Real (in Portuguese). Universidade Nova de Lisboa 318 pp. (1997).
12. Luis Gomes, João Paulo Barros, Using hierarchical structuring mechanisms with Petri nets for PLD based system design. In: *Workshop on Discrete-Event System Design, DESDes'01*, 27–29 June 2001; Zielona Gora, Poland, (2001).
13. Luis Gomes, João Paulo Barros, On structuring mechanisms for Petri nets based system design. In: *Proceedings of ETFA'2003—2003 IEEE Conference on Emerging Technologies and Factory Automation Proceedings*, September, 16–19, 2003, Lisbon, Portugal; IEEE Catalog Number 03TH86961 ISBN 0-7803-7937-3.

14. Luis Gomes, Adolfo Steiger-Garção, Programmable controller design based on a synchronized colored Petri net model and integrating fuzzy reasoning. In: *Application and Theory of Petri Nets'95*; Giorgio De Michelis, Michel Diaz (eds.); Lecture Notes in Computer Science; Vol. 935; Springer, Berlin, pp. 218–237 (1995).
15. David Harel, Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–274 (1987).
16. Xudong He, John A. N. Lee. A methodology for constructing predicate transition net specifications. *Software-Practice and Experience*, **21**(8), 845–875 (August 1991).
17. P. Huber, K. Jensen, R.M. Shapiro, Hierarchies in Coloured Petri Nets. In: *Advances in Petri Nets 1990*, Lecture Nores in Computer Science, Vol. 483, G. Rozenberg (ed.); pp. 313–341 (1990).
18. Kurt Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Vol. 1. Springer-Verlag, ISBN 3-540-55597-8 (1992).
19. Manuel Silva, *Las Redes de Petri: En la Automática y la Informática*. Editorial AC, Madrid, ISBN 84 7288 045 1 (1985).

Chapter 14

IMPLEMENTING A PETRI NET SPECIFICATION IN A FPGA USING VHDL

Enrique Soto[1] and Miguel Pereira[2]
[1]Dept. Tecnología Electrónica, Universidad de Vigo, Apdo. Oficial, 36200 Vigo, España, esoto@uvigo.es
[2]Intelsis Sistemas Inteligentes S.A.—R&D Digital Systems Department, Vía Edison 16 Polígono del Tambre 15890, Santiago de Compostela (La Coruña): e-mail mpereira@intelsis.es

Abstract: This paper discusses how the FPGA architectures affect the implementation of Petri net specifications. Taking into consideration the observations from that study, a method is developed for obtaining VHDL descriptions amenable to synthesis, and tested against other standard methods of implementation. These results have relevance in the integration of access technologies to high-speed telecommunication networks, where FPGAs are excellent implementation platforms.

Key words: Petri nets; VHDL; FPGA; synthesis.

1. INTRODUCTION

In many applications it is necessary to develop control systems based on Petri nets[1]. When a complex system is going to be implemented in a small space, the best solution may be to use a FPGA.

FPGA architectures[2] are divided in many programmable and configurable modules that can be interconnected with the aim of optimizing the use of the device surface. It is necessary to remember that the main problem of PLDs, PALs, and PLAs is the poor use of the device surface, that is, the low percentage of logic gates used. This occurs because this kind of programmable device has only one matrix for AND operations and another matrix for OR operations. FPGAs are different because they are composed of small configurable logic blocks (CLBs) that work like sequential systems. CLBs are composed of a RAM memory and one or more macrocells. Each CLB RAM memory is programmed with the combinational system that defines the behavior of the sequential system.

On each macrocell a memory element (bistable) and a configuration circuit are included. The configuration circuit defines the behavior of the macrocell.

VHDL is a standard hardware description language capable of representing the hardware independent of its final implementation[3]. It is also widely supported by a number of simulation and synthesis tools.

2. FPGA AND PETRI NETS

It is necessary to take into account the following points for implementing a sequential system on a FPGA:

- The system must be divided into low complexity subsystems for integrating them on each CLB of the FPGA.
- Usually, CLBs have only four or five inputs, and one or two outputs (macrocells), and sometimes it is necessary to achieve a strong division of the global system for integrating the subsystems into the CLBs.
- CLBs are interconnected through buses. These buses are connected to configurable connection matrices that have a limited capacity. It is necessary to bring near one another subsystems with a strong dependence between them to optimize the use of these connection matrices.

These points can be followed in many cases when implementing a Petri net. There are two kinds of elements in a Petri net: places and transitions. The circuit implementation of these elements is relatively easy, as shown in the schematics of a place and a transition in Fig. 14-1. Each one of these elements can be programmed in one or more CLBs following the model shown in Fig. 14-1. That would not be the most compact and efficient design, but it would be the simplest.

Each place and transition can be implemented on a CLB. The main problem is the low number of inputs in a CLB. Sometimes it is necessary to use more

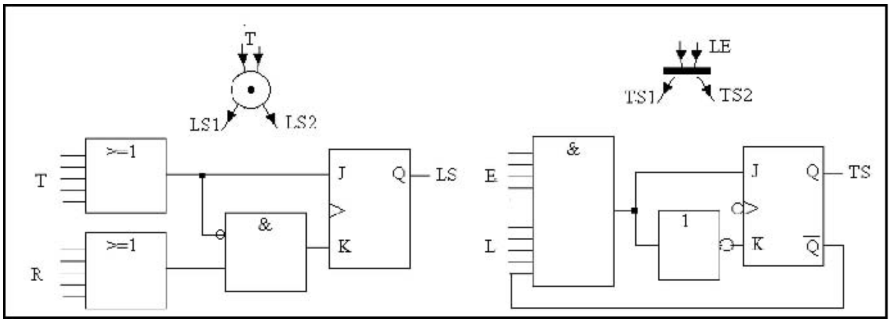


Figure 14-1. Electrical schematics of a place and a transition.

CLBs for each element. T inputs are the signals generated for the preceding transitions. R inputs are connected to the output of the next transitions. LS is the output signal of the place. E inputs are the system inputs involved in the transition. L inputs are the signals generated for the preceding places. TS is the output of the transition.

For obtaining the most compact and efficient design, it is necessary to make the following transformations.

In the models of Fig. 14-1, each place of the Petri net is associated to one bit-state (one-hot encoding). This is not the most compact solution because most of the designs do not need every combination of bits for defining all the states of the system. For instance, in many cases a place being active implies that a set of places will not be active. Coding the place bits with a reduced number of bits will be a good solution because the number of CLBs decreases. For instance, if a Petri net with six places always has only one place active, it is enough to use only 3 bits for coding the active place number (binary state encoding).

The other transformation consists of implementing the combinational circuit of the global system and dividing the final sequential system (combinational circuit and memory elements).

These transformations are used for making compact and fast designs, but they have some limitations.

When a compacted system is divided, maybe too many CLBs have to be used, because of the low number of inputs on each CLB (four or five inputs). This obstacle supposes sometimes to use more CLBs than dividing a noncompacted system.

Verifying or updating a concrete signal of the Petri net in a compacted system may be difficult. It is necessary to take into account the achieved transformations and to supply the inverse transformations for monitoring the signal. This problem can be exposed in failure-tolerant systems. This kind of systems need to verify their signals while they are running. This system may be more complex if it has been compacted previously.

To avoid the mentioned problems, this paper proposes a solution that consists in implementing the system using special blocks composed of one place and a transition. With this kind of blocks compact systems can be achieved, preserving the Petri net structure. Figure 14-2 shows an example of Petri net divided into five blocks. Each block is implemented in a CLB.

3. IMPLEMENTATION

With this kind of implementation of Petri net-based systems, every CLB is composed of a place connected to a transition. The place can be activated

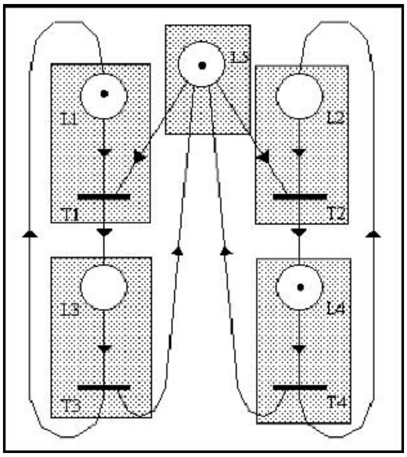


Figure 14-2. Example of Petri net divided into blocks for implementation in FPGA.

through its inputs connected to other transitions. It will be deactivated through reset inputs, or through the transition that is in the block (Fig. 14-3).

The transition will be active when the preceding place is active and the transition inputs have the appropriated values. Every block has two outputs, one state bit corresponding to the place, and a transition bit.

Figure 14-3 presents the description of the new blocks that are developed in a configurable block in a FPGA. On the left, a simplified digital schematic of the block is shown, where

- T is the input bits set connected to other transitions for activating the place,
- R is the vector of signals for deactivating the place since other transitions,
- LE are the signals coming from other places and other input signals that let the activation of the transition,

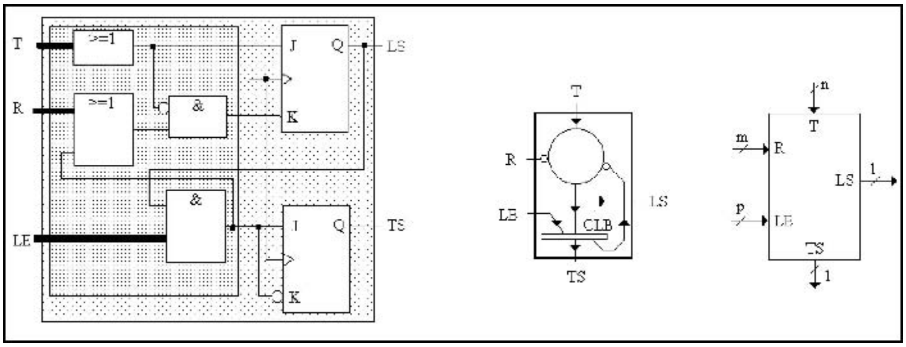


Figure 14-3. Description of the new blocks.

- LS is the output place, and
- TS is the output transition,

Figure 14-3 shows logical and electronic schematics of these blocks. The place and the transition are interconnected through two signals in the block. These signals are not always connected to the exterior. This detail allows a reduction of the CLB connections in a FPGA. In many cases a concrete CLB does not have enough inputs for including a block of this kind. In those cases, it is necessary to use auxiliary CLBs for implementing the block. However, it is unusual to find a Petri net on which every place is preceded by a high number of transitions (or to find a transition preceded by many places). Usually, most places and transitions in a practical Petri net are connected to one or two transitions or places, respectively (except common resources or synchronism points). Figures 14-4 and 14-5 show some examples in which there is an element preceded by many others.

There are cases in which the number of CLB inputs is not enough to include a place or a transition in the CLB. Figure 14-6 shows a logical schematic for expanding the block inputs. The logic gates connected outside the block place-transition are used for incrementing the number of inputs. In this figure, four CLBs are necessary for implementing the block. Three of them are auxiliary blocks and have the function of concentrating a number of inputs in one signal.

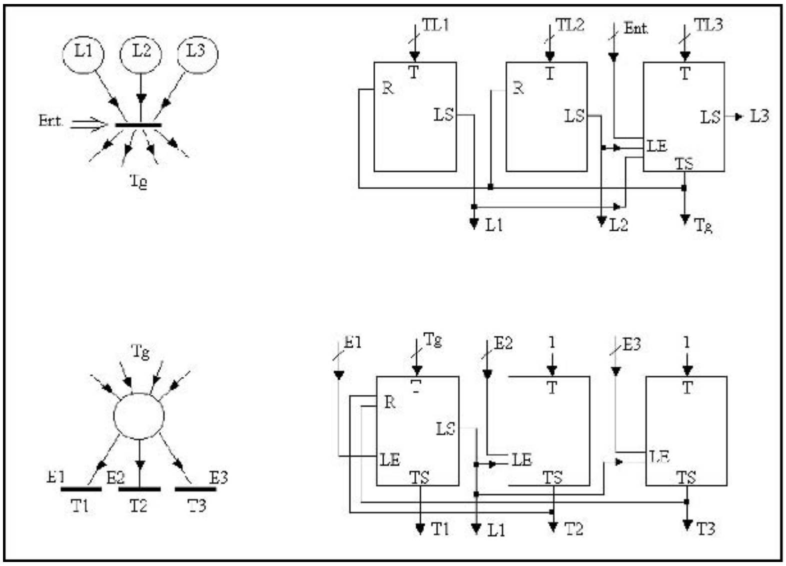


Figure 14-4. Examples of different block interconnections for implementing several places (above) or several transitions (below) with one other element.

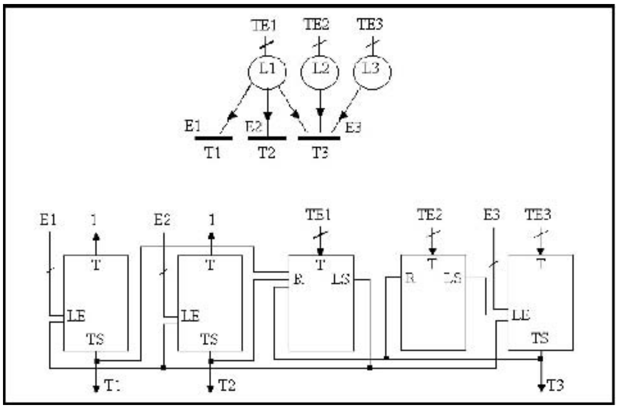


Figure 14-5. Example of interconnection for implementing several places and transitions.

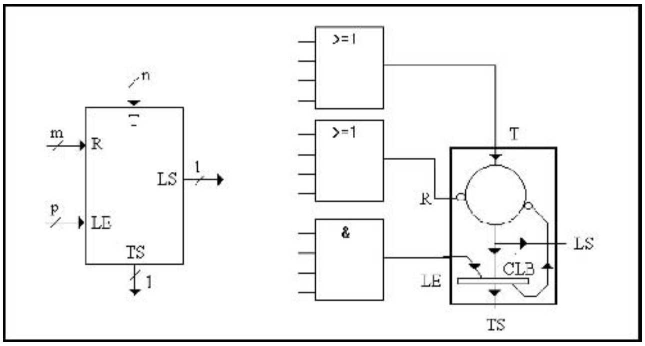


Figure 14-6. Block schematic with a high number of inputs.

4. VHDL HIGH LEVEL DESIGN

The methodology proposed uses the blocks described above as a set of parametrizable objects available in VHDL libraries. The implementation is simply the interconnection, according to the Petri net specification, of those objects whose correctness is guaranteed. The VHDL description represents correctly the specification as long the Petri net does it. The resulting architecture that is implemented within the FPGA is OHE (one-hot encoding).

This solution gives best results in the implementation of SRAM-based FPGA[4], at least as long as the number of places and the random logic associated with the transitions is not too complex relative to the combinational logic available in the FPGA.

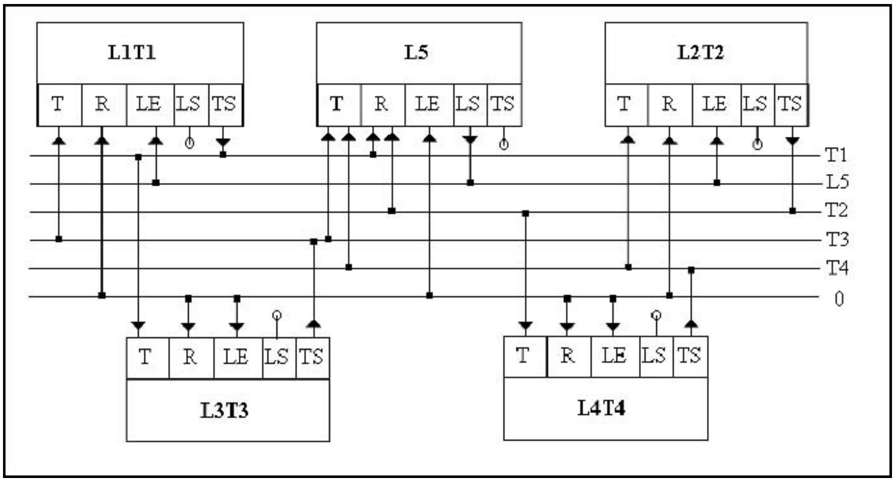


Figure 14-7. Schematic of the connections for the Petri net of Fig. 14-1 with configurable blocks.

5. EXAMPLE

Figure 14-7 shows the blocks interconnection of a Petri net-based system on a FPGA. The given example corresponds to the net of Fig. 14-2.

Each block is a CLB of the FPGA, and it is not necessary to include auxiliary blocks for incrementing the number of inputs of the elements of the net. There is only a place with two input signals in the net of Fig. 14-2. Rest of the places have only one input signal. If some element had more than two inputs, it would be necessary to use the structures of Fig. 14-4, and then the number of CLBs would be increased. The results of different design methodologies using a sample FPGA are summarized in Table 14-1.

Table 14-1. The results of the FPGA design methods

Design method	Design process	FPGA resources in use	Device frequency achieved
Schematic	Difficult	17%	27, 62 MHz
VDHL behavioral	Simple	21%	16, 75 MHz
This paper	Simple	12%	63, 69 MHz

6. CONCLUSIONS

In this paper, the implementation of Petri net-based systems on FPGAs has been discussed. The main problem consists of using places and transitions with a different number of inputs, including the case when there are more inputs than a configurable block of a FPGA. For that, a method has been developed through two circuit models, one for places and the other for transitions. With these models a new block has been presented that contains a place interconnected with a transition. The purpose of this block is to reduce the interconnections between CLBs in a FPGA and, therefore, reducing the number of inputs on each block (especially, feedback signals necessary to reset preceding places). This method is optimal for Petri nets in which most places and transitions are preceded by one or two (but no more) transitions or places. Furthermore, some possibilities have been shown for the interconnection of blocks that increase the number of inputs in elements of a Petri net. The main purpose of this method is to integrate the maximum number of elements of a Petri net in a FPGA.

ACKNOWLEDGMENTS

This work was financed by the European Commission and the Comisión Interministerial de Ciencia y Tecnología (Spain) through research grant TIC 1FD97-2248-C02-02 in collaboration with Versaware SL (Vigo, Spain).

REFERENCES

1. R. Zurawski, M.C. Zhou, Petri nets and industrial applications: A tutorial. *IEEE Transactions on Industrial Electronics* (December 1994).
2. FLEX8000 HandBook ALTERA Corp. (1994).
3. E. Soto, E. Mandado, J. Farina, Lenguajes de descripcion hardware (HDL): El lenguaje VHDL. *Mundo Electronico* (April 1993).
4. John Nemec, Stoke the fires of FPGA design. Keep an FPGA's architecture in mind and produce designs that will yield optimum performance and efficiency. *Electronic Design* (October 25, 1994).

Chapter 15

FINITE STATE MACHINE IMPLEMENTATION IN FPGAs

Hana Kubátová
Department of Computer Science and Engineering, Czech Technical University in Prague, Karlovo nám. 13, 121 35 Prague 2, Czech Republic; e-mail: kubatova@fel.cvut.cz

Abstract: This paper deals with the possibility of the description and decomposition of the finite state machine (FSM). The aim is to obtain better placement of a designed FSM to the selected FPGA. It compares several methods of encoding of the FSM internal states with respect to the space (the number of CLB blocks) and time characteristics. It evaluates the FSM benchmarks and seeks for such qualitative properties to choose the best method for encoding before performing all FOUNDATION CAD system algorithms, since this process is time consuming. The new method for encoding the internal FSM states is presented. All results are verified by experiments.

Key words: finite state machine (FSM); hardware description language (HDL); state transition graph (STG); encoding method; decomposition; benchmark.

1. INTRODUCTION

Most research reports and other materials devoted to searching for the "optimal" encoding of the internal states of a FSM are based on the minimum number of internal states and sometimes also on the minimum number of flip-flops used in their hardware implementation. The only method to get really optimal results is testing of all possibilities[1]. But sometimes "wasting" the internal states or flip-flops is a better solution because of the speed of the designed circuit and mapping to a regular structure. Most encoding methods are not suitable for these structures, e.g., different types of FPGAs or CPLDs. Therefore it is desirable to compare several types of sequential circuit benchmarks to search for the relation between the type of this circuit (the number of the internal states, inputs, outputs, cycles, branching) and the encoding method with respect to their implementation in a *XILINX FPGA*.

Our research group has been working on encoding and decomposition methods for FSMs. We have worked with the CAD system *XILINX FOUNDATION* v2.1i during our first experiments and next with *XILINX ISE*. We have used the benchmarks from the Internet in KISS2 format, some encoding algorithms from *JEDI* program and system *SIS 1.2* [10]. First of all, we classified the FSM benchmarks to know their quantitative characteristics: the number of internal states, inputs, outputs, transitions (i.e., the number of arcs in the state transition graph (STG)), a maximum number of input arcs, a maximum number of output arcs to and from STG nodes, etc. We compared eight encoding methods: "one-hot," "minimum-lengths" (binary), "Johnson," and "Gray" implemented in *FOUNDATION* CAD system. "Fan-in" and "Fan-out" oriented algorithms, the algorithm "FAN" connecting Fan-in and Fan-out ones,[1,5] and the "two-hots" methods were implemented. The second group of our experiments was focused on FSM decompositions. Our original method called a "FEL-code" is based on these experimental results and is presented in this paper. The final results (the number of CLB blocks and maximum frequency) were obtained for the concrete FPGA implementation (Spartan XCS05-PC84).

2. METHODS

2.1 Encoding methods

A one-hot method uses the same number of bits as the number of internal states, where the great number of internal variables is its main disadvantage. The states that have the same next state for the given input should be given adjacent assignments ("Fan-out oriented"). The states that are the next states of the same state should be given adjacent assignments ("Fan-in oriented"). The states that have the same output for a given input should be given adjacent assignments, which will help to cover the 1's in the output Karnaugh maps ("output oriented" method).

A very popular and frequently used method is the "minimum-length" code (obviously called "binary") that uses the minimum number of internal variables, and the Gray code with the same characteristics and adjacent codes for a sequence of the states.

The "two-hots" method uses a combination of two 1's to distinguish between all states.

First partial results based on several encoding methods (Table 15.1) and benchmarks characteristics were presented in Refs. 6, 7, 8, and 9. We have found out that the most successful methods are "minimum-length" and "one-hot." Minimum-length encoding is better than one-hot encoding for small FSMs and for FSMs that fulfill the following condition: a STG describing the FSM

Table 15-1. Examples of codes for six internal states

	Binary	Gray	Johnson	one-hot	two -hot
St1	000	000	000	000001	0011
St2	001	001	001	000010	0101
St3	010	011	011	000100	1001
St4	011	010	111	001000	0110
St5	100	110	110	010000	1010
St6	101	111	100	100000	1100

should be complete or nearly complete. If the ratio of the average output degree of the node to the number of states is greater than 0.7, then it is better to use the minimum-length code. On the contrary, one-hot encoding is better when this ratio is low. Let us define this qualitative property of the FSM as a characteristic *AN*:

$$AN = \frac{\text{Average output edges}}{\text{Number of states} - 1} \tag{1}$$

The value $AN = 0.7$ was experimentally verified on benchmarks and on our specially generated testing FSMs – Moore-type FSMs with the determined number of internal states and the determined number of the transitions from the internal states. Our FSM has the STG with the strictly defined number of edges from all states. For each internal state this number of output edges must be the same. The resulting format is the KISS2 format – e.g., 4.kiss testing FSM has the STG with four edges from each internal state (node). The next state connections were generated randomly to overcome the *XILINX FOUNDATION* optimization for the counter design. The relationship between one-hot and minimum-length encoding methods is illustrated in Fig. 15-1. The

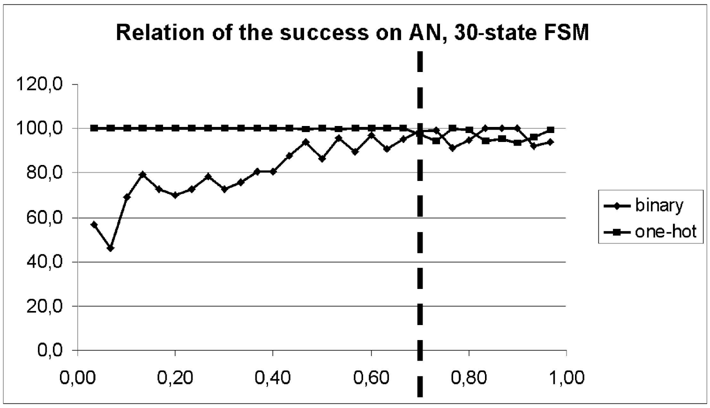


Figure 15-1. The comparison of minimum-length and one-hot encoding with respect to AN.

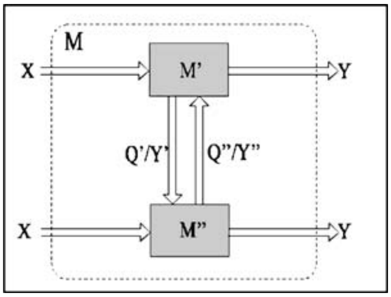


Figure 15-2. Partial decomposition.

comparison was made for 30 internal states FSMs with different branching. The number of transitions from all states is expressed by *AN* (axis X). The axis Y expresses the percentage success of the methods with respect to the space (the minimum number of the CLB from all encoding methods for every testing FSM is 100%). It can be seen that for less branching (smaller *AN*) the one-hot code is always better until the border $AN = 0.7$.

2.2 Decomposition types

Decomposition[1] means splitting of one whole into several (simpler) parts that implement the function of the former whole. We were interested in the decomposition of the FSM into several cooperating sub-FSMs. Decomposition can be *partial* or *full* according to division of only the internal states (partial) or all the sets of input, output, and internal states (full). See Fig. 15-2, where X is the finite set of input symbols, Y is the finite set of output symbols, Q is the finite set of internal states, the FSM $M = (X, Y, Q, \delta, \lambda)$ is to be split into sub-FSMs $M' = (X', Y', Q', \delta', \lambda')$ and $M'' = (X'', Y'', Q'', \delta'', \lambda'')$.

The following decomposition types are classified as *two-level* and *several-level* ones according the number of sub-FSMs. There are decompositions which are distinguished according the way of cooperation: *simultaneous* decomposition, where all sub-FSMs work simultaneously; and *cascade* decomposition, where sub-FSMs work sequentially and where the next sub-FSM starts to work when its predecessor finishes its function. All types of decomposition can be parallel, serial, or general according the type of exchange of information about internal states. The strict definition can be seen in Refs. 1, 2, 3, and 4.

2.3 "FEL-code" method

Our method combines both the one-hot and minimum-length encoding methods. It is based on the partial FSM internal state decomposition, such

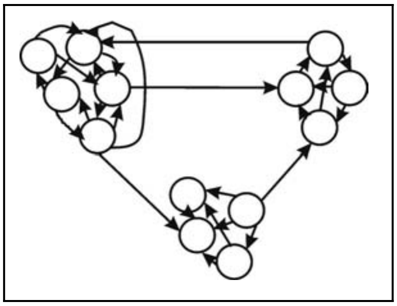


Figure 15-3. FEL-code basis.

as several level, cascade, and general, where the number of levels depends on the FSM properties. The global number of states of the decomposed FSM is equal to the original number of FSM internal states. The internal states are divided into groups (sets of internal states) with many connections between states (for such "strongly connected" states the minimum-length encoding is better to use); see Fig. 15-3. The internal state code is composed of the minimum-length part (a serial number of the state in its set in binary notation) and the one-hot part (a serial number of a set in one-hot notation). The number of minimum-length part bits is equal to b, where 2^b is greater than or equal to the maximum number of the states in sets and the number of one-hot part bits corresponds to the number of sets.

The global algorithm could be described as follows:

1. Place all FSM internal states Q_i into the set S_0.
2. Select the state Q_i (from S_0) with the greatest number of transitions to other disjoint states from S_0. Take away Q_i from S_0; Q_i becomes the first member of the new set S_{group}.
3. Construct the set $S_{neighbor}$ of neighboring internal states of all members of S_{group}. Compute the score expressing the placement suitability for a state Q_j into S_{group} for all states from $S_{neighbor}$. Add the state with the highest score to S_{group}.
4. The score is a sum of
   (a) the number of transitions from Q_j to all states from S_{group} multiplied by the constant 10;
   (b) the number of such states from S_{group} for which exists the transition from Q_j to them, multiplied by the constant 20;
   (c) the number of transitions from Q_j to all neighboring internal states from S_{group} (i.e. to all states from $S_{neighbor}$) multiplied by the constant 3;
   (d) the number of such states from $S_{neighbor}$ for which the transition from Q_j to them, multiplied by the constant 6, exists;

(e) the number of transitions from all internal states from S_{group} to Q_j multiplied by the constant 10;
(f) the number of such states from S_{group} that the transition from them to Q_j, multiplied by the constant 20, exists;
(g) the number of transitions from all neighboring states of S_{group} (placed in $S_{neighbor}$) to Q_j, multiplied by the constant 3;
(h) the number of the neighboring states in $S_{neighbor}$ for which the transition from them to Q_j, multiplied by the constant 6, exists;

5. Compute AN (1) characteristics for S_{group}.
6. When this value is greater than the "border value" (the input parameter of this algorithm, in our experiments is usually 0.7) the state Q_j becomes the real member of S_{group}. Now continue with step 3. When the ratio is less than the border value, state Q_j is discarded from the S_{group} and this set is closed. Now continue with step 2.
7. If all internal states are placed into sets S_i and at the same time S_0 is empty, construct the internal states code:
8. The code consists of the binary part (a serial number of the state in its set in binary notation) and the one-hot part (a serial number of a set in one-hot notation). The number of binary part bits is equal to b, where 2^b is greater than or equal to the maximum number of states in the sets. The number of one-hot part bits is equal to the number of sets S_i.

Example (*lion* benchmark[10], border ratio 0.7, Fig. 15-4)

1. Place all FSM internal states Q_i into the set S_0

   $S_0 = \{\text{st0, st1, st2, st3}\}$
2. For all S_0 elements compute the number of transitions to next disjoint states from S_0:

   (st0 . . . 1, st1 . . . 2, st2 . . . 2, st3 . . . 1)

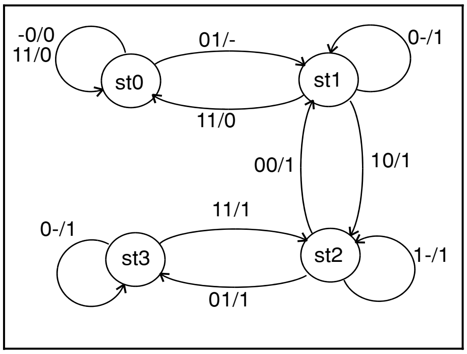


Figure 15-4. STG of the *lion* benchmark.

Choose the state with the highest value and construct the new set S_1:

$S_0 = \{\text{st0, st2, st3}\}, \quad S1 = \{\text{st1}\}$

3. Construct the set $S_{neighbor}$ of neighboring internal states of all members of S_1:

   $S_0 = \{\text{st0, st2, st3}\}, \quad S_1 = \{\text{st1}\}, \quad S_{neighbor} = \{\text{st0, st2}\}$

   Compute the score for all states from $S_{neighbor}$:

   $st0_{score} = 1.10 + 1.20 + 2.3 + 1.6 + 1.10 + 1.20 + 2.3 + 1.6 = 84$
   $st2_{score} = 1.10 + 1.20 + 1.3 + 1.6 + 1.10 + 1.20 + 1.3 + 1.6 = 78$

   Choose the state with the highest score and add it to S_1:

   $S_0 = \{\text{st2, st3}\}, \quad S_1 = \{\text{st0, st1}\}, S_{neighbor} = \{\text{st2}\}$

4. Compute *AN* (1) for the elements from S_1:

   $AN = 1.0.$

   AN is greater then 0.7; therefore, the state Q_j becomes a real member of S_1. Now continue with step 3.

5. Try to add the state *st2* into S_1 and compute *AN*. Because $AN = 0.66$, state *st2* is discarded from S_1 and this set is closed. Now continue by step 2.

At the end all internal states are placed into two groups:

$S_1 = \{\text{st0, st1}\}, \quad S_2 = \{\text{st2, st3}\}$

Now the internal state code is connected from the one-bit binary part and the two-bit one-hot parts:

st0 . . . 0/01 st1 . . . 1/01 st2 . . . 0/10 st3 . . . 1/10

3. EXPERIMENTS

Because the conversion program between a KISS2 format and VHDL was necessary, the converter *K2V_DOS* (in *C++* by the compiler *GCC* for *DOS OS*) was implemented[6]. The *K2V_DOS* program allows obtaining information about FSMs; e.g., the node degree, the number of states, the number of transitions, etc. The VHDL description of a FSM created by the *K2V_DOS* program can be described in different ways (with different results):

- One big process sensitive to both the clock signal and the input signals (one *case* statement is used in this process; it selects an active state; in each branch of the *case* there are *if* statements defining the next states and outputs). This is the same method that was used for the conversion between STG and VHDL by the *XILINX FOUNDATION*[11].
- Three processes (*next-state-proc* for the implementation of the next-state function, *state-dff-proc* for the asynchronous reset and D flip-flops application, and *output-proc* for the FSM output function implementation). To

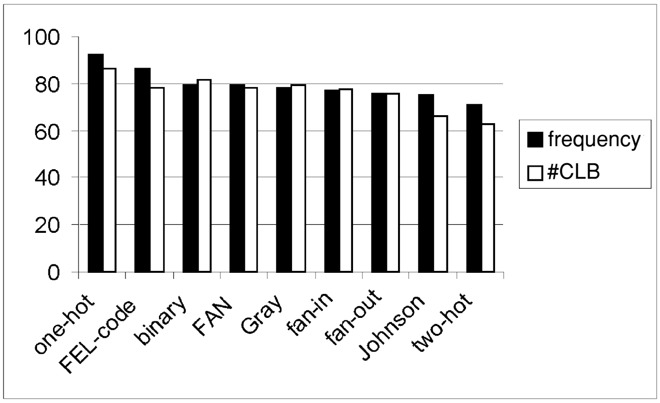


Figure 15-5. The final comparison of all encoding methods used.

overcome the *XILINX FOUNDATION* optimization for the one-hot encoding method, the direct code assignment was used as well.

The *K2V_DOS* program system can generate our special testing FSMs (for more precise setting of the characteristics *AN*, see Section 2.1). The *K2V_DOS* program can generate different FSM internal state encoding by minimum-lengths, Gray, Johnson, one-hot, two-hots, Fan-in, Fan-out, FAN, and FEL-code methods. All benchmarks were processed by the *DECOMP* program to generate all possible types of decompositions (in KISS2 format due to using the same batch for the FPGA implementation).

4. RESULTS AND CONCLUSIONS

We have performed about 1000 experiments with different types of encoding and decomposition methods for 50 benchmarks. The comparison of nine encoding methods is shown in Fig. 15-5. There is condensed information about the average "success" of all encoding methods. The minimum number of CLB blocks for a benchmark is divided by the number of CLB blocks for a particular encoding method. Similarly, the working frequency for a particular encoding method is divided by the maximum frequency for a benchmark. These values were computed for all benchmarks for maximum working frequency (dark columns) and number of CLB blocks (#CLB, white columns) and expressed as percentage "success" for each encoding method.

We can reach the following conclusions based on our encoding and decomposition experiments:

- The minimum-length encoding method provides the best results for FSM with a few internal states (5) and for FSM with *AN* > 0.7 (the state transition graph with many cycles).

Table 15-2. Frequency improvement

	Average maximum frequency (MHz)		
Encoding method	Spartan	Spartan-II	Frequency increase (%)
Binary	76.6	145.0	**47.2**
one-hot	80.5	167.1	**51.8**
FEL-c 0.5	71.3	108.7	**34.4**
FEL-c 0.7	75.6	128.1	**41.0**

- One-hot encoding method is better for other cases and mostly generates faster circuits (but the *XILINX FOUNDATION* uses optimization methods for "one-hot" encoding).
- The FEL-code method is universal as it combines the advantages of both one-hot and minimum-length methods. This method is heuristic; the parameters (*AN* and the score evaluations) have been experimentally verified.
- Other tested encoding methods provide worse results in most cases and have no practical significance.
- For such FSM implementation where majority of CLB blocks are used (e.g., 90%), the one-hot method gives better results, mainly with respect to the maximum working frequency due to easier wiring.
- All FSM decomposition types are not advantageous to use in most cases due to great information exchange – the parallel decomposition is the best one (if it exists).
- A different strategy for searching for the partitions – the best FSM partition is not the one with the minimum number of internal states but the one with the minimum sets of input and output symbols – could be used for FPGA implementation.

The experimental results performed on the recent *XILINX ISE* CAD system have not been sufficiently compared with those presented above, since many qualitative changes were incorporated into this tool, such as new types of final platforms and new design algorithms. According to our last results not yet presented, we can conclude that the one-hot and minimum-length methods still remain the most successful. The average improvement (for all benchmarks but only for working frequency) is presented in Table 15-2. It can be stated that the encoding methods offered by the CAD system are better then the outside methods and AN = 0.7 is a right value.

ACKNOWLEDGMENTS

This paper is based on the results of several student projects supervised by the author for four years. This research was in part supported by the grants "Design of Highly Reliable Control Systems Built on Dynamically Reconfigurable FPGAs" (GA 102/04/2137) and "Modern Methods of Digital Systems Design" (GA 102/04/0737).

REFERENCES

1. P. Ashar, F. Devadas, A.R. Newton, *Sequential Logic Synthesis*. Kluwer Academic Publishers, Boston, Dordrecht, London (1992).
2. L. Józwiak, J.C. Kolsteren, An efficient method for sequential general decomposition of sequential machines. *Microprocessing and Microprogramming*, **32** 657–664 (1991).
3. A. Chojnaci, L. Jozwiak, An effective and efficient method for functional decomposition of Boolean functions based on information relationship Measures. In: *Proceedings of 3rd DDECS 2000 Workshop*, Smolenice, Slovakia, pp. 242–249 (2000).
4. L. Jozwiak, An efficient heuristic method for state assignment of large sequential machines. *Journal of Circuits, Systems and Computers*, **2** (1) 1–26 (1991).
5. K. Feske, S. Mulka, M. Koegst, G. Elst, Technology-driven FSM partitioning for synthesis of large sequential circuits targeting lookup-table based FPGAs. In: *Proceedings 7th Workshop Field-Programable Logic and Applications (FPL '97)*, Lecture Notes in Computer Science, **1304**, London, UK, pp. 235–244 (1997).
6. H. Kubátová, T. Hrdý, M. Prokeš, Problems with the Enencoding of the FSM with the Relation to its Implementation by FPGA. In: *ECI2000 Electronic Computers and Informatics, International Scientific Conference, Herl'any*, pp. 183–188 (2000).
7. H. Kubátová, Implementation of the FSM into FPGA. In: *Proceedings of the International Workshop on Discrete-Event System Design DESDes '01*, Oficyna Wydawnicza Politechnika Zielona Góra, Przytok, Poland, pp. 141–146 (2001).
8. H. Kubátová, How to obtain better implementation of the FSM in FPGA. In: *Proceedings of the 5th IEEE Workshop DDECS 2002*, Brno, Czech Republic, pp. 332–335 (2002).
9. H. Kubátová, M. Bečvář, FEL-Code: FSM internal state enencoding method. In: *Proceedings of 5th International Workshop on Boolean Problems*. Technische Universität Bergakademie, Freiberg, pp. 109–114 (2002).
10. ftp://ftp.mcnc.org/pub/benchmark/Benchmark.dirs/LGSynth93/LGSynth93.tar
11. The Programmable Logic Data Book. XILINX Tenth Anniversary (1999), http://www.xilinx.com

Chapter 16

BLOCK SYNTHESIS OF COMBINATIONAL CIRCUITS

Pyotr Bibilo and Natalia Kirienko
United Institute of Informatics Problems of the National Academy of Sciences of Belarus, Logic Design Laboratory, Surganov Str. 6, 220012 Minsk, Belarus;
e-mail: Bibilo@newman.bas-net.by Kir@newman.bas-net.by

Abstract: Circuit realization on a single programmable logic array (PLA) may be unacceptable because of the large number of terms in sum-of-products; therefore a problem of block synthesis is considered in this paper. This problem is to realize a multilevel form of Boolean function system by some blocks, where each block is a PLA of smaller size. A problem of block synthesis in gate array library basis is also discussed in this paper. The results of experimental research of influence of previous partitioning of Boolean function systems on circuit complexity in PLA and gate array library basis are presented in this paper.

Key words: synthesis of combinational circuits; partitioning; programmable logic array (PLA); gate array library; layout.

1. INTRODUCTION

There are different ways of implementation of the control logic of custom digital VLSI circuits. The most important ways are realization of two-level AND/OR circuits in programmable logic array (PLA) basis[9] and realization of multilevel circuits in library gates basis[7]. Each of them has its advantages and disadvantages. The advantage of PLA circuits is simplicity of layout design, testing, and modification, because the circuits are regular. There are effective methods and programs of PLA area minimization[9,4]. The disadvantage of two-level PLA circuits is the large chip area in comparison with the area required for a multilevel library gates circuit. But the synthesis of a multilevel circuit is a very difficult task[5]; moreover, such circuits are harder for testing and topological design than PLA circuits. The implementation of a circuit in a single PLA may be unacceptable because of the large size of the PLA. Therefore a problem of block synthesis is considered in this paper. The results of experimental research are given.

2. REPRESENTATION OF BOOLEAN FUNCTIONS AND THE BASIS OF SYNTHESIS

It is well known that the formal (mathematical) model of functioning of a multioutput combinational circuit is a system of completely defined Boolean functions[9]. Let a combinational circuit have n inputs and m outputs. One of the forms of Boolean function system representation is the sum-of-product (SOP) system. Let us denote by $D_f(n, k, m)$ the system of Boolean functions $f(x) = (f^1(x), \dots, f^m(x))$, $x = (x_1, \dots, x_n)$, specified on k common elementary products of Boolean variables $x_1, \dots, x_n$. Let us represent $D_f(n, k, m)$ by a pair of matrices, the ternary $k \times n$ matrix T^x and the Boolean $k \times m$ matrix B^f. A pair of the appropriate rows of T^x and B^f represents, respectively, the product and the subset of the functions that belongs to their SOPs. The representation of a system of Boolean functions in the form of a SOP system will be called a *two-level representation.* We refer to the specification of a system of Boolean functions in the form of a system of parenthesized algebraic expression on the basis of logic AND, OR, and NOT operators as the multilevel representation.

A programmable logic array is a classical two-level structure for realization of a SOP system of Boolean functions. A system of SOP $D_f(n, k, m)$ can be realized on PLA (n, m, k), which has not less than n input pins, m output pins, and k intermediate lines. Elementary products are realized on intermediate lines of matrix AND of PLA, whereas SOPs are realized in matrix OR of PLA. The PLA structure is adequate for the system of SOPs $D_f(n, k, m)$. The commutation points between input pins and intermediate lines in AND matrix correspond to fixed (0,1) elements of T^x, and the commutation points between output pins and intermediate lines in OR matrix correspond to elements 1 of B^f.

The library gates used as basis elements for synthesis of combinational logic circuits are the elements from the logic gate library K1574[3]. Each element of such a library is characterized by the number of basis gates needed for its location in the chip. There are various elements in the gate library K1574: inverters, multiplexers, buffer elements, and gates AND, OR, NAND, NOR, XOR etc.

3. ALGORITHM FOR PARTITIONING OF MULTILEVEL REPRESENTATION OF A SYSTEM OF BOOLEAN FUNCTIONS

Let us consider the multilevel algebraic form of a Boolean function system in AND/OR/NOT basis. Each intermediate or output function is given by a separate SOP. Let nonoverlapping SOP subsets $R_1, \dots, R_h$ form a partition of SOP set $D = \{D^1, \dots, D^m\}$. $R_1, \dots, R_h$ are the blocks of the partition with the following parameters: n_i is the number of input variables, m_i is the number of output variables, and k_i is the number of products in two-level form of SOP of functions of the block.

Let the restrictions (n^*, m^*, k^*) imposed on block complexity be given, where n^* is the maximal number of input variables, m^* is the maximal number of output variables, and k^* is the maximal number of products in two-level representation of SOP of functions of the block.

Transformation of multilevel into two-level representation and determination of the parameter k_i is related to solving problems of intermediate variables elimination[6] and joint minimization of a system of Boolean functions in SOP form[4,5,9].

The problem of partitioning of multilevel representation of a system of Boolean functions is to find a partition $R_1, \dots, R_h$ that fulfils the (n^*, m^*, k^*) restriction and has a minimum number of blocks h.

The main idea of the technique for solving this problem is the following. The blocks (or SOP subsystems) build up step by step. A SOP with the maximal number of external variables is chosen out as the starting point for formation of the next subsystem. Then SOPs that are most closely connected to this subsystem (having maximal number of common variables) are added to it, and the elimination of intermediate variables and joint minimization of functions in SOP subsystem are performed. The resulting subsystem is checked for the fulfillment of the (n^*, m^*, k^*) restriction. If the (n^*, m^*, k^*) restriction is not violated, then the next SOP is added to this subsystem; otherwise the constructing of the next subsystem begins. As a result, each SOP is in one of the subsystems. This algorithm was described in detail in Ref. 2.

4. BLOCK METHOD FOR SYNTHESIS IN PLA BASIS

The block method for synthesis in PLA basis has two procedures. The first procedure is the partitioning of multilevel representation of the system of Boolean functions into blocks with (n^*, m^*, k^*)-restricted parameters. The second procedure is realization of each block by one PLA.

Let the area of a circuit consisting of some (possibly interconnected) PLAs be equal to the sum of the areas of these PLAs. The conditional area of one PLA (n, m, k), evaluated in conditional units (bits), is determined by the formula

$$S_{PLA}^{log} = (2n + m)k(bit). \tag{1}$$

At the level of a particular layout used in silicon compiler SCAS[1], the real PLA area is determined by the formula

$$S_{PLA}^{top} = S_{AND,OR} + S_{BND}, \tag{2}$$

where $S_{AND,OR}$ is the area of information matrices *AND, OR*, determined by the formula

$$S_{AND,OR} = 2 * \left]\frac{k}{8}\right[* \left(9 * \left]\frac{n}{2}\right[* + 10 * \left]\frac{m}{4}\right[\right), \tag{3}$$

and S_{BND} is the area of the PLA boundary (load transistors, buffers, etc.), determined by the formula

$$\begin{aligned} S_{BND} = 34{,}992 * \left]\frac{k}{8}\right[+ 85 * \left]\frac{n}{2}\right[+ 49{,}104 * \left]\frac{m}{4}\right[\\ + \left(\left]\frac{m}{4}\right[- 1\right) * 12{,}276 + 60{,}423. \end{aligned} \tag{4}$$

The value of S_{PLA}^{top} determined by formulas (3) and (4) is the number of real layout cells that the PLA layout is composed of[1].

5. BLOCK METHOD FOR SYNTHESIS IN GATE ARRAY LIBRARY BASIS

The block method for synthesis in gate array library basis has two procedures. The first procedure is the partitioning of multilevel representation of the system of Boolean functions into blocks with (n^*, m^*, k^*)-restricted parameters. The second procedure is realization of technology mapping of each block into the logic gate library. This procedure includes synthesis in the gate array library using a basic method, “Cover.”

The basic method “Cover” is a process of covering Boolean expressions in AND/OR/NOT basis by elements from the gate library. Previously, each multiplace AND or OR operator of the system is replaced by superposition of two-place AND or OR operators, respectively. Then the Boolean network is built for each expression, where each node corresponds to two-place operator AND/OR or one-place operator NOT. The problem of covering of Boolean network is to find subnetworks in it, which are functionally equivalent to library elements. The basic method "Cover" was described in detail in Ref. 3. It is experimentally confirmed to be better than the method represented in Ref. 7.

6. EXPERIMENTAL INVESTIGATION OF BLOCK METHODS FOR SYNTHESIS IN PLA AND GATE ARRAY LIBRARY BASE

The block methods for synthesis were implemented in computer programs and investigated experimentally. The experiments were done on a series of combinational circuits from the well-known MCNC benchmark set chosen from design practice. The programs run on PC Celeron 600, RAM 64 Mb.

Experiment 1. Two realizations of multilevel representation in PLAs were compared: the first is realization in single PLA; the second is realization in several PLAs, obtained by the partition algorithm. Table 16-1 shows the results of Experiment 1.

Experiment 2. Three realizations of multilevel representation in the basis of logic gate library were compared: the basis method "Cover," the combining method, and the block method of synthesis. The combining method is composed of two steps. The first step is transformation of multilevel representation into two-level representation. The second step is synthesis by basis method "Cover." Table 16-2 shows the results of Experiment 2.

The minimization of two-level representations of Boolean function system in partition algorithm was performed by the program of joint minimization in SOP[8]. The basis method "Cover" uses the computer program from Ref. 3.

The notation in Tables 16-1 and 16-2 is as follows:

n, the number of arguments of the realized Boolean function system (the number of input pins in the circuit);

m, the number of functions in the system (the number of output pins in the circuit);

k, the number of products in the SOP system (two-level representation);

$S_{PLA}^{\log}$, the conditional area of one PLA (n, m, k), evaluated in conditional units (bits) by formula (1);

$\sum S_{PLA}^{\log}$, the sum of conditional areas of PLAs, found by the block synthesis method;

S_{PLA}^{top}, the real area of one PLA (n, m, k), evaluated by formula (2);

$\sum S_{PLA}^{top}$, the sum of real areas of PLAs, found by the block synthesis method;

S, the circuit complexity in library gate basis (the total number of gates required for logical elements, i.e. area of elements);

Table 16-1. Comparison of two realizations of multilevel representation for PLAs: the first is realization in one PLA; the second is realization in several PLAs, obtained by partition algorithm

Circuit				One PLA		PLA net			
name	n	m	k	S^{log}_{PLA}	S^{top}_{PLA}	n^*, m^*, k^*	h	$\sum S^{log}_{PLA}$	$\sum S^{top}_{PLA}$
x1	**51**	**35**	**274**	**37538**	**26715,29**	**40, 20, 500**	**3**	*24521*	*21982, 5*
apex6	**132**	**94**	**432**	**154656**	**99092,84**	**100, 70, 900**	**3**	**104864**	**73608, 43**
						105, 80, 900	**2**	**113252**	**79032, 04**
						70, 40, 900	**4**	*75872*	*60442, 43*
apex7	**49**	**35**	**213**	**28329**	**20680,35**	**30, 30, 500**	**3**	**17104**	**16603, 77**
						35, 30, 500	**2**	**18666**	**16008, 26**
						25, 20, 500	**4**	*8825*	*10526, 71*
example2	**85**	**63**	**161**	**37513**	**28394,06**	**40, 30, 900**	**4**	*19501*	**20406, 34**
						60, 40, 900	**2**	**23450**	**20429, 21**
						55, 35, 900	**2**	**21227**	*19194, 59*
x4	**94**	**71**	**371**	**95347**	**63474,61**	**60, 35, 900**	**4**	*40638*	*35639, 54*
						66, 50, 900	**3**	**46910**	**46237, 44**
frg2	**143**	**139**	**3090**	**1313250**	**794310,4**	**50, 30, 500**	**11**	*171581*	*166857, 8*
too_large	**38**	**3**	**1021**	*80659*	*52539,5*	**50, 2, 900**	**4**	**102170**	**73014, 45**
ttt2	**24**	**21**	**222**	**15318**	**11824,2**	**24, 10, 900**	**3**	**10773**	**13958, 77**
						24, 8, 900	**5**	*8092*	*9907, 832*
cm150a	**21**	**1**	**796**	**34228**	**26343,73**	**16, 2, 500**	**3**	*6155*	*6422, 373*
						18, 3, 500	**2**	**11623**	**14899, 51**
frg1	**28**	**3**	**119**	**7021**	*5904,407*	**26, 1, 100**	**3**	*5783*	**6441, 453**
						28, 2, 100	**2**	**6523**	**8530, 519**
lal	**26**	**19**	**117**	**8307**	*6994,927*	**30, 16, 100**	**5**	*4574*	**8248, 276**
add8	**17**	**9**	**2519**	**108317**	**81949,77**	**12, 9, 900**	**2**	*9324*	*9184, 462*
x3	**135**	**99**	**915**	**337635**	**209646,7**	**80, 50, 900**	**5**	*170755*	*122437, 9*
term1	**34**	**10**	**818**	**63804**	**42979,46**	**30, 8, 500**	**2**	**49896**	**56050, 37**
						34, 4, 500	**3**	**36750**	**27895, 76**
						20, 8, 500	**8**	*21327*	*23514, 69*
mux	**21**	**1**	**425**	**18275**	**14706,1**	**16, 10, 100**	**2**	**1862**	*3025, 974*
						12, 6, 100	**3**	*1519*	**3270, 493**
Count	**35**	**16**	**89**	**7654**	**7091,571**	**25, 8, 100**	**4**	*2769*	*5072, 029*
						18, 6, 100	**6**	**4851**	**5606, 067**

L, the number of logical elements in a circuit;
n^*, partition parameter (the number of arguments of a block);
m^*, partition parameter (the number of functions in a block);
k^*, partition parameter (the number of products in a block); and
h, the number of blocks in the partition of multilevel representation of Boolean functions system.

According to the results of the experiment the following conclusions can be stated.

Table 16-2. Comparison of basis method, combining method, and block realization of multi-level representation

			Basis method		Combining method			Block realization		
Circuit name	*n*	*m*	*L*	*S*	*k*	*L*	*S*	*h*	*L*	*S*
cu	**14**	**11**	**41**	*204*	**35**	**76**	**397**	**4**	**41**	**219**
comp	**32**	**3**	**110**	**555**	**3**	**12**	*70*	**2**	**23**	**123**
cmb	**16**	**4**	**27**	*142*	**56**	**116**	**630**	**3**	**38**	**212**
cm162a	**14**	**5**	**43**	**198**	**20**	**57**	**290**	**2**	**24**	*117*
mux	**21**	**1**	**61**	*319*	**425**	**1032**	**6094**	**5**	**61**	**327**
count	**35**	**16**	**111**	**506**	**89**	**188**	**1024**	**13**	**76**	*356*
frg1	**28**	**3**	**298**	**1683**	**119**	**298**	**1683**	**3**	**298**	*1683*
cm138a	**6**	**8**	**9**	**53**	**6**	**16**	**88**	**3**	**9**	*53*
cm82a	**5**	**3**	**20**	**90**	**11**	**14**	**74**	**3**	**13**	*57*
9symml	**9**	**1**	**154**	**738**	**30**	**70**	*359*	**5**	**141**	**727**
lal	**26**	**19**	**117**	*576*	**117**	**207**	**1141**	**3**	**180**	**849**
unreg	**36**	**16**	**96**	**432**	**49**	**80**	**384**	**2**	**80**	*384*
z4ml	**7**	**4**	**102**	**554**	**19**	**26**	*140*	**2**	**102**	**554**
x3	**135**	**99**	**933**	*4475*	**915**	**1948**	**10808**	**4**	**966**	**4679**
pcle	**19**	**9**	**39**	**181**	**16**	**40**	**214**	**5**	**36**	*173*
term1	**34**	**10**	**495**	**2407**	**818**	**2414**	**13403**	**7**	**465**	*2360*
cm150a	**21**	**1**	**61**	*261*	**796**	**2136**	**11886**	**4**	**65**	**309**
too_larg	**38**	**3**	**5217**	**30145**	**1027**	**5273**	**30473**	**4**	**4915**	*28392*
ttt2	**24**	**21**	**299**	**1609**	**222**	**561**	**2937**	**5**	**225**	*1197*
sct	**19**	**15**	**120**	**584**	**64**	**124**	**618**	**6**	**127**	*583*
c8	**28**	**18**	**159**	**798**	**70**	**92**	**464**	**4**	**92**	*452*
frg2	**143**	**139**	**1315**	*6560*	**3090**	**15385**	**85093**	**15**	**1361**	**6886**
cm42a	**4**	**10**	**13**	*65*	**4**	**20**	**94**	**3**	**14**	**72**

The block method for synthesis of multilevel representation by a PLA net is preferable to single PLA realization. The gain for area is obtained in 13 circuits from 16. The parameters of circuits with the smallest area are given in bold (see Table 16-1). Only macroelement area was taken into account in this experiment, and the bound area was not. Thus the final conclusion about replacement of one PLA with PLA net can be made after layout design. Using formula (2) for area calculation is preferable to using formula, (1). For example, area calculation for Frg1, Lal with (1), gives advantage, but the real PLA net area is more than the single PLA area.

The block realization is preferable for synthesis in logic gate library too. The better (minimum) valuations of circuit complexity are given in bold (Table 16-2). The transformation of multilevel representation into two-level representation (combining method) is advisable in only three examples; it is not competitive with the basis method "Cover" and the block method.

REFERENCES

1. P.N. Bibilo, Symbolic layout of VLSI array macros with an SCAS silicon compiler. I. *Russian Microelectronics*, **27** (2) 109–117 (1998).
2. P.N. Bibilo, N.A. Kirienko, Partitioning a system of logic equations into subsystem under given restrictions. In: *Proceedings of the Third International Conference on Computer-Aided Design of Discrete Devices (CAD DD'99)*, Vol. 1, Republic of Belarus, Minsk, pp. 122–126 (1999).
3. P.N. Bibilo, V.G. Litskevich, *Boolean Network Covering by Library Elements. Control Systems and Computers*, Vol. 6, Kiev, Ukraine, pp. 16–24 (1999) (in Russian).
4. K.R. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithm for VLSI Synthesis*. Kluwer Academic Publisher, Boston (1984).
5. K.R. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli, A.R. Wang, MIS: A multiply-level logic optimisation systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **CAD-6** 1062–1081 (1987).
6. L.D. Cheremisinova, V.G. Litskevich, *Realization of Operations on Systems of Logic Equations and SOPs. Logical Design*. Institute of Engineering Cybernetics of National Academy of Sciences of Belarus, Minsk, pp. 139–145 (1998) (in Russian).
7. F. Mailhot, G. De Micheli, Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **12** (5) 599–620 (1993).
8. N.R. Toropov, *Minimization of Systems of Boolean Functions in DNF. Logical Design*. Institute of Engineering Cybernetics of National Academy of Sciences of Belarus, Minsk, pp. 4–19 (1999) (in Russian).
9. A.D. Zakrevskij, *Logic Synthesis of the Cascade Circuits*. Nauka, Moscow (1981) (in Russian).

Chapter 17

THE INFLUENCE OF FUNCTIONAL DECOMPOSITION ON MODERN DIGITAL DESIGN PROCESS

Mariusz Rawski[1], Tadeusz Łuba[1], Zbigniew Jachna[2], and Paweł Tomaszewicz[1]
[1]Warsaw University of Technology, Institute of Telecommunications, Nowowiejska 15/19, Warsaw, Poland; rawski@tele.pw.edu.pl, luba@tele.pw.edu.pl, ptomasze@tele.pw.edu.pl
[2]Military University of Technology, Kaliskiego 2, Warsaw, Poland

Abstract: General functional decomposition has been gaining more and more importance in recent years. Though it is mainly perceived as a method of logic synthesis for the implementation of Boolean functions into FPGA-based architectures, it has found applications in many other fields of modern engineering and science. In this paper, an application of balanced functional decomposition in different tasks of modern digital designing is presented. The experimental results prove that functional decomposition as a method of synthesis can help implementing circuits in CPLD/FPGA architectures. It can also be efficiently used as a method for implementing FSMs in FPGAs with Embedded ROM Memory Blocks.

Key words: Logic Synthesis; Functional Decomposition; FPGA; FSM; ROM.

1. INTRODUCTION

Decomposition has become an important activity in the analysis and design of digital systems. It is fundamental for many fields of modern engineering and science[1,2,3,4]. The functional decomposition relies on breaking down a complex system into a network of smaller and relatively independent cooperating subsystems, in such a way that the behavior of the original system is preserved, i.e., function F is decomposed to subfunction G and H in the form described by formula $F = H(A, G(B))$.

New methods of logic synthesis based on the functional decomposition were recently developed[5,6,7]. One of the promising decomposition-based methods is the so-called balanced decomposition[8].

Since multilevel functional decomposition yields very good results in the logic synthesis of combinational circuits, it is viewed mostly as a synthesis method for the implementation of combinational functions into FPGA-based architectures[9,10]. However, a decomposition-based method can be used beyond this field. In the sequential machine synthesis after the state code assignment of the process of implementation is reduced to the computation of a flip-flop excitation function, the decomposition can be efficiently used to assist such an implementation. An application of a balanced decomposition method allows the designer to decide what the optimization criterion is—is it circuit area or circuit speed. Good results produced by the decomposition-based logic synthesis methods in implementation of combinational circuits guarantee that this method will implement encoded sequential machines efficiently and effectively. The balanced decomposition gives the designer control over the process of excitation function implementation. Therefore, such undesirable effects as hazards can be avoided. Elimination of these effects can increase the speed of circuits.

Modern FPGA architectures contain embedded memory blocks. In many cases, designers do not need to use these resources. However, such memory blocks allow implementing sequential machines in a way that requires less logic cells than the traditional, flip-flop implementation. This may be used to implement "nonvital" sequential parts of the design saving logic cell resources for more important parts. However such an implementation may require more memory than is available in a circuit. To reduce memory usage in ROM-based sequential machine implementations decomposition-based methods can be successfully used[11].

In this paper some basic information has been introduced and the application of the balanced decomposition in the implementation of combinational parts of digital systems has been discussed. The application of the decomposition in the implementation of sequential machines is also presented. Subsequently, some experimental results, reached with a prototype tool that implements the balanced functional decomposition has also been discussed.

The experimental results demonstrate that balanced decomposition is capable of constructing solutions of comparable or even better quality than the methods implemented in a university or in commercial systems.

2. BALANCED FUNCTIONAL DECOMPOSITION

Some preliminaries necessary for understanding this paper are presented here. More detailed information concerning balanced functional decomposition method can be found in Ref. 8, 12.

Balanced decomposition relies on the partitioning of a switching function with either parallel decomposition or serial decomposition applied at each phase

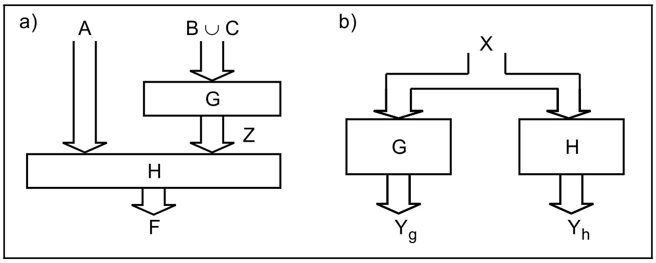


Figure 17-1. Schematic representation of a) the serial and b) the parallel decomposition.

of the synthesis process. In parallel decomposition, the set of output variables Y of a multioutput function F is partitioned into subsets, Y_g and Y_h, and the corresponding functions, G and H, are derived so that, for either of these two functions, the input support contains less variables than the set of input variables X of the original function F (Fig. 17-1b). An objective of parallel decomposition is to minimize the input support of G and H.

In serial decomposition, the set of input variables X is partitioned into subsets, A and B, and functions G and H are derived so that the set of input variables of G is $B \cup C$, where C is a subset of A, the set of input variables of H is $A \cup Z$, where Z is the set of output variables of G, and H has less input variables than the original function F, i.e. $F = H(A, G(B, C))$ (Fig. 17-1a).

The balanced decomposition is an iterative process in which, at each step, either parallel or serial decomposition of a selected component is performed. The process is carried out until all resulting subfunctions are small enough to fit blocks with a given number of input variables.

The idea of intertwining parallel and serial decomposition has been implemented in a program called DEMAIN. This tool is designed to aid implementation of combinational parts of digital systems and this application has two modes: automatic and interactive. It can also be used for the reduction of the number of inputs of a function, when an output depends on only a subset of the inputs. From this point of view DEMAIN is a tool specially dedicated to FPGA-oriented technology mapping.

Example: The influence of the balanced decomposition on the final result of the FPGA-based mapping process will be explained with the function F (Table 17-1) describing one of benchmark examples with 10 input variables and 2 output variables, for which cells with 4 inputs and 1 output are assumed (this is the size of Altera's FLEX FPGA)[13].

As F is a ten-input, two-output function, in the first step of the decomposition both the parallel and serial decomposition can be applied. Let us apply parallel

Table 17-1. Truth table of function F in espresso format

.type fr	1100010001 00	0001001011 11
.i 10	0011101110 01	1110001110 10
.o 2	0001001110 01	0011001011 10
.p 25	0110000110 01	0010011010 01
0101000000 00	1110110010 10	1010110010 00
1110100100 00	0111100000 00	0100110101 11
0010110000 10	0100011011 00	0001111010 00
0101001000 10	0010111010 01	1101100100 10
1110101101 01	0110001110 00	1001110111 11
0100010101 01	0110110111 11	.e

decomposition at first. Parallel decomposition generates two components with 6 inputs and one output each. Assuming that the inputs of the primary function are denoted as 0, 1, . . . ,9 (from the left column to the right column), and the outputs are y_0 and y_1, the inputs of the obtained components are 0, 1, 3, 4, 6, 7 and 0, 1, 2, 6, 7, 9, respectively. Each of the above components is subject to two-stage serial decomposition.

For the first stage the nondisjoint serial decomposition can be applied, as shown on the left hand side of Figure 17-2. The second component can also be decomposed serially, however with the number of outputs of the extracted block G equal to two. Therefore, minimizing the total number of decomposition components, a disjoint decomposition strategy is applied, resulting in components G_2 and H_2, as shown on the right side of Fig. 17-2. The truth tables of the components of function F produced by decomposition algorithm DEMAIN (denoted G_1, H_1, G_2 and H_2 in Fig. 17-2) are shown in Tables 17-2a, 17-2b, 17-2c and 17-2d, respectively. Thus it is clear that the function F can be mapped onto four logic cells of Altera's FLEX structure.

It is worth noting, that the same function synthesised directly by commercial tools e.g., MAX+PlusII, Quartus and Leonardo Spectrum is mapped onto 35, 29, and 95 logic cells, respectively.

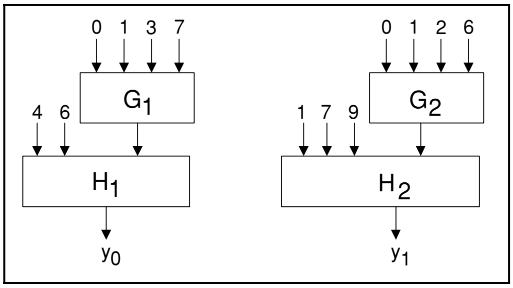


Figure 17-2. Decomposition of function F.

Table 17-2. Truth tables of decomposition blocks

a) G_1	b) H_1	c) G_2	d) H_2
0110 1	–01 0	0110 1	10–1 0
1101 1	011 1	0011 1	–101 1
1000 1	111 0	0100 1	–111 1
0010 1	100 1	1000 1	0011 0
0000 0	0–0 0	0101 1	0001 1
0101 0	110 0	1100 0	1–00 0
1100 0		0010 0	0000 0
0100 0		1010 0	1110 1
0011 0		1110 0	1010 0
1011 0		0001 0	0100 1
1111 0		0111 0	0010 1
		1111 0	

3. FINITE STATE MACHINE IMPLEMENTATION

FSM can be implemented using ROM (*Read Only Memory*)[11]. In the general architecture of such an implementation state and input variables ($q_1, q_2, \ldots, q_n$ and $x_1, x_2, \ldots, x_m$) constitute ROM address variables ($a_1, a_2, \ldots, a_{m+n}$). The ROM would consist of words, each storing the encoded present state (control field) and output values (information field). The next state would be determined by the input values and the present-state information feedback from memory.

This kind of implementation requires much less logic cells than the traditional flip-flop implementation (or does not require them at all, if memory can be controlled by clock signal—no address register required); therefore, it can be used to implement "nonvital" FSMs of the design, saving LC resources for more important sections of the design. However, a large FSM may require too many memory resources.

The size of the memory needed for such an implementation depends on the length of the address and the memory word.

Let m be the number of inputs, n be the number of state encoding bits and y be the number of output functions of FSM. The size of memory needed for implementation of such an FSM can be expressed by the following formula:

$$M = 2^{(m+n)} \times (n + y),$$

where $m + n$ is the size of the address, and $n + y$ is the size of the memory word.

Since modern programmable devices contain embedded memory blocks, there exists a possibility of implementing FSM using these blocks. The size of the memory blocks available in programmable devices is limited. For example,

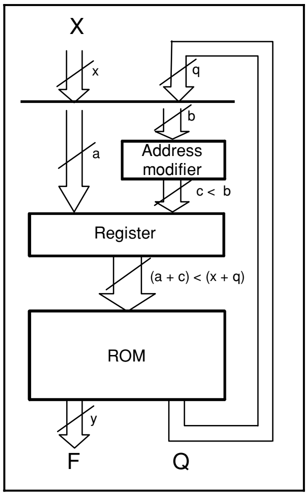


Figure 17-3. Implementation of FSM using an address modifier.

Altera's FLEX family EAB (*Embedded Array Block*) has 2048 bits of memory and the device FLEX10K10 consists of 3 such EABs. Functional decomposition can be used to implement FSMs that exceed that size.

Any FSM defined by a given transition table can be implemented as in Fig. 17-3, using an address modifier. The process may be considered as a decomposition of memory block into two blocks: a combinational address modifier and a smaller memory block. Appropriately chosen strategy of the balanced decomposition may allow reducing required memory size at the cost of additional logic cells for address modifier implementation. This makes possible the implemention of FSM that exceeds available memory through using embedded memory blocks and additional programmable logic.

Example: FSM implementation with the use of the concept of address modifier.

Table 17-3. a) FSM table, b) state and input encoding

a)

	v1	v2	v3	v4
s1	s1	s2	s4	–
s2	–	–	s5	s4
s3	s3	s2	s1	s3
s4	s2	–	s4	s1
s5	s3	s1	s4	s2

b)

	x1x2					
	00	01	10	11	Q2 q3	q1
	v1	v2	v3	v4		
s1	s1	s2	s4	–	00	0
s2	–	–	s5	s4	01	
s4	s2	–	s4	s1	10	
s3	s3	s2	s1	s3	11	1
s5	s3	s1	s4	s2	01	

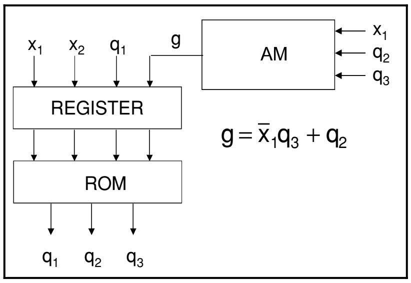


Figure 17-4. Implementation of FSM using an address modifier.

Let us consider FSM described in Table 17-3a. Its outputs are omitted, as they do not have influence on the method. This FSM can be implemented using ROM memory with 5 addressing bits. This would require memory of the size of 32 words. In order to implement this FSM machine in ROM with 4 addressing bits, the address modifier is required.

Let us implement the given FSM in a structure shown on Fig. 17-4 with 3 free variables and one output variable from the address modifier. Such an implementation requires memory of the size of 16 words and additional logic to implement the address modifier.

To find the appropriate state/input encoding and partitioning, the FSMs state transition table is divided into 8 subtables (encoded by free variables), each of them having no more then two different next states, which can be encoded with one variable—address modifier output variable (to achieve this, rows s_3 and s_4 changed places with each other). Next the appropriate state and input encoding is introduced (Table 17-3b).

The truth table of address modifier, as well as content of the ROM memory, can be computed by the application of the serial functional decomposition method.

More detailed description of the method can be found in Ref. 11.

4. EXPERIMENTAL RESULTS

The balanced decomposition was applied to implement FPGA in architectures of several "real life" examples: combinational functions and combinational parts of FSMs. We used the following examples:

- bin2bcd1—binary to BCD converter for binary values from 0 to 99,
- bin2bcd2—binary to BCD converter for binary values from 0 to 355,
- rd88—sbox from Rijndael implementation,
- DESaut—combinational part of the state machine used in DES algorithm implementation,

Table 17-4. Implementation of real-life examples

Example	DEMAIN	MAX+Plus II	QuartusII	FPGA Express	Leonardo Spectrum
bin2bcd1	13	165	38	30	30
bin2bcd2	39	505	393	225	120
rd88	262	326	452	–	–
DESaut	28	46	35	25	30
5B6B	41	92	41	100	49
count4	11	74	68	17	11

- 5B6B—the combinational part of the 5B-6B coder,
- count4—4 bit counter with COUNT UP, COUNT DOWN, HOLD, CLEAR and LOAD.

For the comparison the following synthesis tools were used: MAX+PlusII v10.2 Baseline, QuartusII WebEdition v3.0 SP1, FPGA Express 3.5, Leonardo Spectrum v1999.1 and DEMAIN. Logic network produced by all synthesis tools were implemented in EPF10K10LC84-3, the FPGA device from FLEX family of Altera.

Table 17-4 shows the comparison of our method based on balanced decomposition as implemented in tool DEMAIN with other methods of compared tools. The table presents the comparison of logic cells needed for implementation of given examples. The results of implementation in FPGA architecture show that the method based on the balanced decomposition provide better results than other tools used in the comparison. It is especially noticeable in synthesis of such parts of digital systems that can be represented by truth table description, as in the case of BIN2BCD converter. In the implementation of this example obtained with DEMAIN software the number of logic cells required are over 10 times less than in the case of the MAX+PlusII and 2 times in case of the Leonardo Spectrum. It is also much better than implementation based on behavioral description[14], which requires 41 logic cells of device EPF10K10. This is over 3 times worse than the solution obtained with DEMAIN.

The presented results lead to the conclusion that the influence of the balanced decomposition on efficiency of practical digital systems implementation would be particularly significant when the designed circuit contains complex combinational blocks. This is a typical situation when implementing cryptographic algorithms, where so-called substitution boxes are usually implemented as combinational logic.

DEMAIN has been used in the implementation of such algorithms allowing significant improvement in logic resources utilization, as well as in performance. Implementation of data path of the DES (*Data Encryption Standard*) algorithm with MAX+PlusII requires 710 logic cells and allows encrypting data with throughput of 115 Mb/s. Application of balanced functional decomposition in optimization of selected parts of the algorithm reduces the number of required logic cells to 296 without performance degradation and even increasing it to 206 Mbits/s[14].

The balanced functional decomposition was also used in the implementation process of the Rijndael algorithm targeted to low-cost Altera programmable devices[15]. Application of DEMAIN software allowed implementing this algorithm in FLEX10K200 circuit very efficiently with throughput of 752 Mbits/s. For comparison, implementation of Rijndael in the same programmable structure developed at TSI (France) and Technical University of Kosice (Slovakia) allowed throughput of 451 Mbits/s, at George Mason University (USA)—316 Mbits/s, and at Military University of Technology (Poland)—248 Mbits/s[16].

Since, upon the encoding of the FSM states, the implementation of such FSM architectures involves the technology mapping of the combinational part into target architecture, the quality of such an implementation strongly depends on the combinational function implementation quality.

In Table 17-5 the comparison of different FSM implementations is presented. Each sequential machine was described by a transition table with encoded states. We present here the number of logic cells and memory bits required (i.e., area of the circuit) and the maximal frequency of clock signal (i.e., speed of the circuit) for each method of FSM implementation.

The columns under the FF_MAX+PlusII heading present results obtained by the Altera MAX+PlusII system in a classical flip-flop implementation of FSM. The columns under FF_DEMAIN heading show results of implementation of the transition table with the use of balanced decomposition. The ROM columns provide the results of ROM implementation; the columns under AM_ROM heading present the results of ROM implementation with the use of an address modifier. It can be easily noticed that the application of balanced

Table 17-5. Implementation of FSM: 1) FSM described with special AHDL construction, 2) decomposition with the minimum number of logic levels, 3) decomposition not possible, 4) not enough memory to implement the project

	FF_MAX+PlusII		FF_DEMAIN		ROM		AM_ROM	
Example	LCs/ Bits	Speed [MHz]	LCs/ Bits	Speed [MHz]	LCs/ Bits	Speed [MHz]	LCs/ Bits	Speed [MHz]
DESaut	46/0	41,1	28/0	61,7	8/1792	47,8	7/896	47,1
5B6B	93/0	48,7	43/0	114,9	6/448	48,0	–3)	–3)
Count4	72/0 18/0 1)	44,2 86,2 1)	11/0 13/0 2)	68,5 90,0 2)	16/16384	–4)	12/1024	39,5

decomposition can improve the quality of flip-flop as well as ROM implementation. Especially interesting is the implementation of the 4-bit counter. Its description with a transition table leads to a strongly nonoptimal implementation. On the other hand, its description when using a special Altera HDL (*Hardware Description Language*) construction produces very good results. However, utilization of balanced decomposition allows the designer to choose between whether area or speed is optimized. The ROM implementation of this example requires too many memory bits (the size of required memory block exceeds the available memory), thus it cannot be implemented in a given structure. Application of functional decomposition allows reducing of the necessary size of memory, which makes implementation possible.

5. CONCLUSIONS

Balanced decomposition produces very good results in combinational function implementation in FPGA-based architectures. However, results presented in this paper show that balanced functional decomposition can be efficiently and effectively applied beyond the implementation of combinational circuits in FPGAs. Presented results, achieved using different algorithms of multilevel synthesis, show that the possibilities of decomposition-based multilevel synthesis are not fully explored.

Implementation of sequential machines in FPGA-based architectures through balanced decomposition produces devices that are not only smaller (less logic cells utilised) but are also often more important, faster than those obtained by the commercial MAX+PlusII tool. Balanced decomposition can also be used to implement large FSM in an alternative way—using ROM. This kind of implementation requires much less logic cells than the traditional flip-flop implementation; therefore, it can be used to implement "nonvital" FSMs of the design, saving logic cell resources for more important parts of the circuits. However, large FSM may require too much memory resources. With the concept of address modifier, memory usage can be significantly reduced.

The experimental results shown in this paper demonstrate that the synthesis method based on functional decomposition can help in implementing sequential machines using flip-flops, as well as ROM memory.

Application of this method allows significant improvement in logic resources utilization as well as in performance. Implementation of data path of the DES (*Data Encryption Standard*) algorithm with MAX+PlusII requires 710 logic cells and allows encrypting data with throughput of 115 Mbits/s. Application of balanced decomposition reduces the number of logic cells to 296 without performance degradation and even increasing it to 206 Mbits/s.

These features make balanced decomposition the universal method that can be successfully used in digital circuits design with FPGAs.

ACKNOWLEDGMENT

This paper was supported by the Polish State Committee for Scientific Research financial grant for years 2003–2004, number 4 T11D 014 24.

REFERENCES

1. J.A. Brzozowski, T. Łuba, Decomposition of Boolean functions specified by cubes. *Journal of Multiple-Valued Logic and Soft Computing*, **9**, 377–417 (2003). Old City Publishing, Inc., Philadelphia.
2. C. Scholl, *Functional Decomposition with Application to FPGA Synthesis*. Kluwer Academic Publishers, 2001.
3. T. Łuba, Multi-level logic synthesis based on decomposition. *Microprocessors and Microsystems*, **18** (8), 429–437 (1994).
4. R. Rzechowski, L. Jóźwiak, T. Łuba, Technology driven multilevel logic synthesis based on functional decomposition into gates. In: *EUROMICRO'99 Conference*, Milan, pp. 368–375 (1999).
5. M. Burns, M. Perkowski, L. Jóźwiak, An efficient approach to decomposition of multi-output Boolean functions with large set of bound variables. *EUROMICRO'98 Conference*, Vol. 1, Vasteras, pp. 16–23 (1998).
6. S.C. Chang, M. Marek-Sadowska, T.T. Hwang, Technology mapping for TLU FPGAs based on decomposition of binary decision diagrams. *IEEE Transactions on CAD*, **15** (10), 1226–1236 (October, 1996).
7. L. Jóźwiak, A. Chojnacki, Functional decomposition based on information relationship measures extremely effective and efficient for symmetric Functions. *EUROMICRO'99 Conference*, Vol. 1, Milan, pp. 150–159 (1999).
8. T. Łuba, H. Selvaraj, M. Nowicka, A. Kraśniewski, Balanced multilevel decomposition and its applications in FPGA-based synthesis, In: G. Saucier, A. Mignotte (eds.), *Logic and Architecture Synthesis*. Chapman & Hall (1995).
9. M. Nowicka, T. Łuba, M. Rawski, FPGA-based decomposition of Boolean functions. Algorithms and implementation. In: *Proceedings of Sixth International Conference on Advanced Computer Systems*, Szczecin, pp. 502–509.
10. M. Rawski, L. Jóźwiak, T. Łuba, Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures. *Journal of Systems Architecture*, **47**, 137–155 (2001).
11. M. Rawski, T. Łuba, FSM Implementation in embedded memory blocks using concept of decomposition. In: W. Ciazynski, et al. (eds.): *Programmable Devices and Systems*. Pergamon—Elsevier Science, pp. 291–296 (2002).
12. T. Łuba, H. Selvaraj, A general approach to Boolean function decomposition and its applications in FPGA-based synthesis, VLSI design. *Special Issue on Decompositions in VLSI Design*, **3** (3–4), 289–300 (1995).
13. http://www.altera.com/products/devices/dev-index.jsp.

14. T. Łuba, M. Rawski, P. Tomaszewicz, B. Zbierzchowski, *Digital Systems Design*, WKŁ, Warsaw (2003) (in Polish).
15. M. Obarski, Implementation of Rijndael cryptographic algorithm in FPGA structure. MSc Thesis, WUT (2004) (in Polish).
16. V. Fischer, M. Drutarovsky, Two methods of Rijndael implementation in reconfigurable hardware. In: *CHES 2001*, pp. 77–92.

Section V

System Engineering for Embedded Systems

Chapter 18

DEVELOPMENT OF EMBEDDED SYSTEMS USING OORT

A case study

Sérgio Lopes, Carlos Silva, Adriano Tavares and João Monteiro
Department of Industrial Electronics, University of Minho, Campus de Azurém, 4800-058 Guimarães, Portugal; e-mail: sergio.lopes@del.uminho.pt, csilva@del.uminho.pt, atavares@del.uminho.pt, joao.monteiro@del.uminho.pt

Abstract: The development of embedded systems requires both tools and methods which help the designer to deal with the higher complexity and tougher constraints due to the different hardware support, often distributed topology and time requirements. Proper tools and methods have a major impact on the overall costs and final product quality. We have applied the Object-Oriented Real-Time Techniques (OORT) method, which is oriented toward the specification of distributed real-time systems, to the implementation of the Multiple Lift System (MLS) case study. The method is based on the UML, SDL and MSC languages and supported by the *Object*GEODE* toolset. This paper summarizes the method and presents our experience in the MLS system development, namely the difficulties we had and the success we have achieved.

Key words: Embedded Systems Specification; Software Engineering; Discrete-Event Systems Control; Simulation; Targeting.

1. INTRODUCTION

Embedded systems are very complex because they are often distributed, run in different platforms, have temporal constraints, etc. Their development demands high quality and increasing economic constraints, therefore it is necessary to minimize their errors and maintenance costs, and deliver them within short deadlines.

To achieve these goals it is necessary to verify a few conditions: decrease the complexity through hierarchical and graphical modeling for high flexibility

**Object*GEODE is a registered trademark by Verilog.

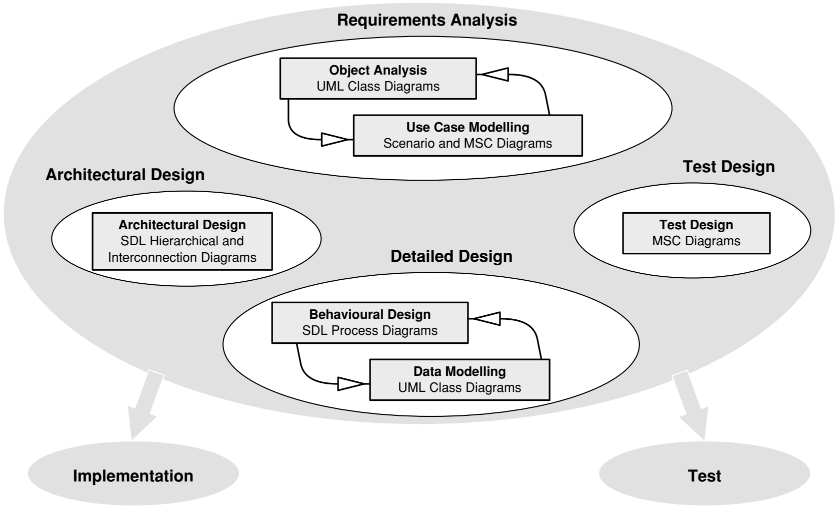


Figure 18-1. The OORT method.

in the maintenance; protect the investments with the application of international standards in the development; apply early verification and validation techniques to reduce the errors; and, reduce the delivery times by automating code generation and increasing the level of reusability. Finally, it is necessary to have a tool that provides functionalities to fulfill them.

The present work was developed with the *Object*GEODE toolset, which supports the OORT method[1], described by the diagram of Fig. 18-1. This method applies the Unified Modeling Language[2] (UML), the Message Sequence Chart[3] (MSC) and the Specification and Description Language[4–6] (SDL). The UML is an international standard defined by the Object Management Group (OMG), for which there are plenty of introductory texts[7]. The MSC and SDL, both international standards by the International Telecommunication Union (ITU), were defined for a combined usage[8]. More information on these languages is available, namely a tutorial[9] for MSC, and for SDL, a handy summary[10], and a more comprehensive reference[11].

In this work the OORT method is applied to a development case study—the Multiple Lift System (MLS). The analysis description uses UML to model the system's environment, and MSC to specify the intended behavior of the system. The system's architecture is defined in SDL. The detailed design applies SDL and UML to the specification of concurrent objects and passive components, respectively. The MSC language supports the test design activity. The simulation of the designed system was carried out with the *Object*GEODE simulator. Finally, the targeting operation was performed with the help of the tools' C Code Generator. The following sections describe each of these steps in the systems engineering process.

2. REQUIREMENTS ANALYSIS

In the requirements analysis phase, the system environment is modeled and the user requirements are specified. The analyst must concentrate on **what** the system should do. The environment where the system will operate is described by means of UML class diagrams—**object modeling**. The functional behavior of the system is specified by MSCs, organized in a hierarchy of scenarios—**use case modeling**. The system is viewed from the exterior as a black box, with which external entities (system actors) interact. Both the object model and the use case model must be independent from the solutions chosen to implement the system in the next phases.

2.1 Object Modeling

In the description of the system environment, the class diagrams are used to express the application and service domains. This is accomplished by identifying the relevant entities of the application domain (physical and logical), their attributes, and the relationships between them. For the sake of clarity, the entities and their relationships should be grouped in modules reflecting different perspectives, as defended by Yourdon[12].

Fig. 18-2 gives an overview of the system environment, where the system's main actors are identified, in this case `Passenger, Potential`

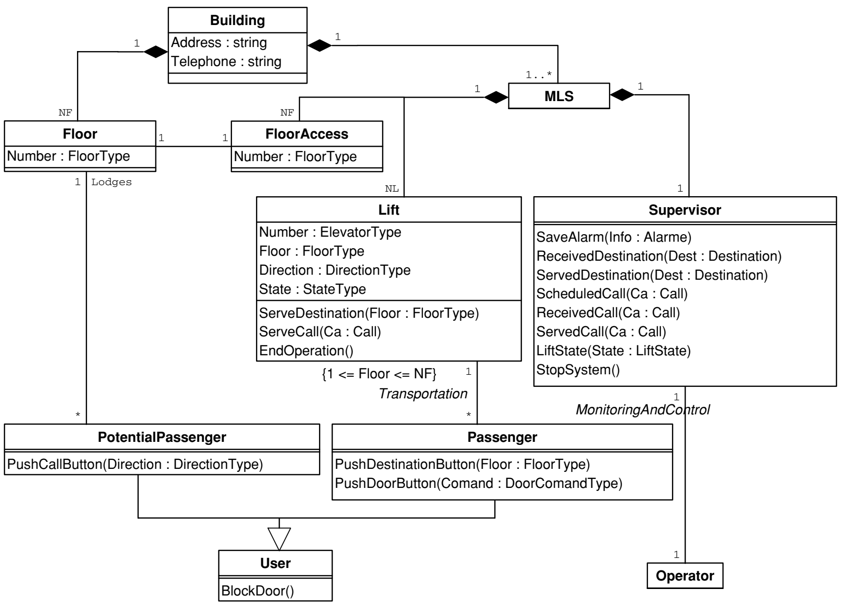


Figure 18-2. UML Class Diagram of the Building Module.

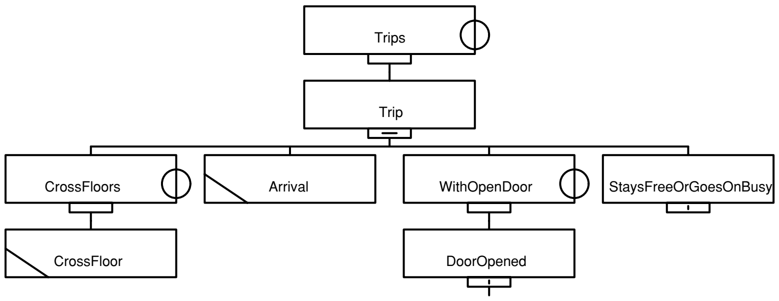


Figure 18-3. MSC Scenario Hierarchy for the Trip Subscenario.

`Passenger` and `Operator`. Generally, there is one module for some basic system composition, one for each of the actors and others to express interesting relationships. More information about the analysis of the MLS can be found in Douglass[13].

2.2 Use Case Modeling

The use case model is composed of a scenario hierarchical tree with MSC diagrams as the leaves. The scenario hierarchy should contain all the different expected scenarios of interaction between the system and its environment. The goal is to model the functional and dynamic requirements of the system. First, the main scenarios are identified, and then are individually refined into more detailed subscenarios, until the terminal scenarios can be easily described by a chronological sequence of interactions between the system and its environment.

This approach faces the problem of a possible scenario explosion. To deal with it, the first step is to make use of the composition operators to hierarchically combine the different scenarios. However, the problem is diminished but not completely solved. It is necessary to make a good choice of scenarios, namely to choose those which are the most representative of the system behavior.

The system operation is divided into phases which are organized by the composition operators, and each phase is a branch in the scenario hierarchy. Fig. 18-3 shows the `Trip` phase scenario hierarchy, in which we have a `CrossFloor` terminal scenario as illustrated in Fig. 18-4.

A constant concern must be the coherence between the use case and the object models[1].

3. ARCHITECTURAL DESIGN

In this phase, the system designers specify a logical architecture for the system (as opposed to a physical architecture). The SDL language covers all aspects of the architecture design.

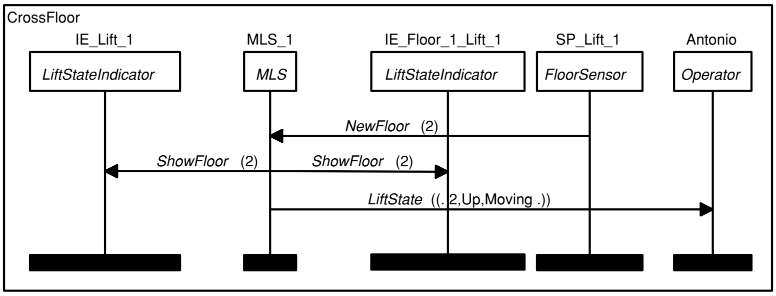


Figure 18-4. Abstract MSC for the `CrossFloor` Scenario.

The system is composed of **concurrent objects** (those which have an execution thread) and **passive objects** (those which implement a set of functions invoked by concurrent objects). The concurrent objects are identified and organized in an architecture hierarchy. This is accomplished by a combination of refinement and composition. The refinement is a top-down process in which higher level objects are divided into smaller and more detailed lower level objects, bearing in mind the modularity aspects.

The composition is a bottom-up process in which designers try to group objects in such a way that it favors reutilization, and pays special attention to encapsulation. Fig. 18-5 illustrates the SDL object's hierarchy for the MLS.

In the architectural design, the real characteristics of the environment where the system will operate should be considered, as well as the efficiency aspects. On the other hand, the SDL model should be independent of the real object distribution on the final platform. At the first level, the system actors are considered through their interfaces, and modeled as channels between the system's top-level objects and the outside world. Fig. 18-6 shows the top level of the MLS architecture.

Some passive objects are also defined, such as signals with complex arguments, Abstract Data Types (ADTs) associated with internal signal processing, and operators to implement the I/O communication with the outside world (instead of signals).

The use of SDL assures the portability of the system architecture. Since the communication is independent of the real object distribution, the channels are dynamic, and the objects can be parameterized.

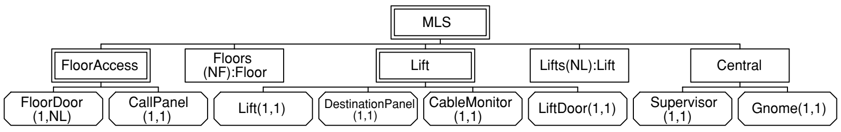


Figure 18-5. SDL Hierarchy Diagram for the MLS.

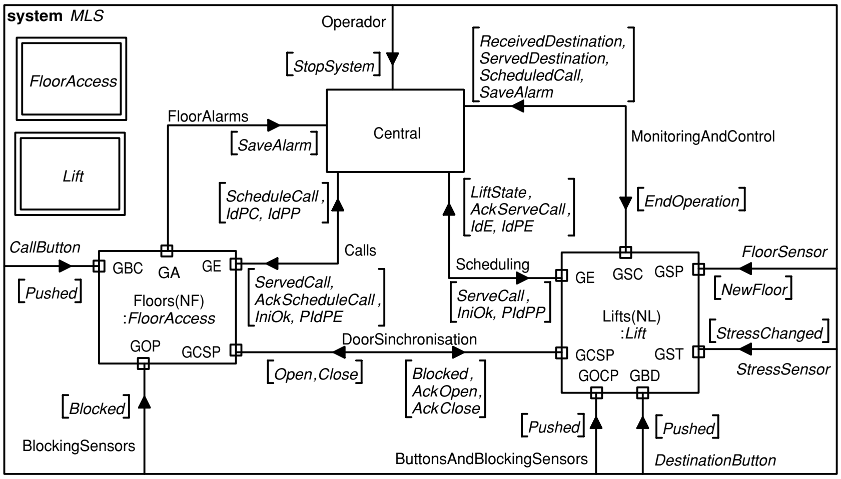


Figure 18-6. SDL Interconnection Diagram of the Top Level MLS Hierarchy.

4. DETAILED DESIGN

The description of concurrent and passive objects that constitute the system architecture is done in the detailed design phase. In other words, it is specified **how** the system implements the expected services, which should be independent of the final platform where the system will run.

4.1 Concurrent Objects Design

The concurrent objects are the terminal objects of the SDL hierarchy. Each one of them is an SDL process described as a kind of finite state machine called **process diagram**. The process diagrams are built by analyzing the input signals for each process, and how the answer to those signals depends on the previous states. The SDL has a set of mechanisms to describe the transitions that allow a complete specification of the process behavior. In Fig. 18-7 the process diagram of the `FloorDoor` process is depicted. The reuse of external concurrent objects is supported by the SDL encapsulation and inheritance mechanisms.

4.2 Passive Objects Design

Some passive objects are identified during the analysis phase. Generally they model data used or produced by the system, and they are included in the detailed design to provide services for the concurrent objects. There are also passive objects that result from design options, such as data management, user

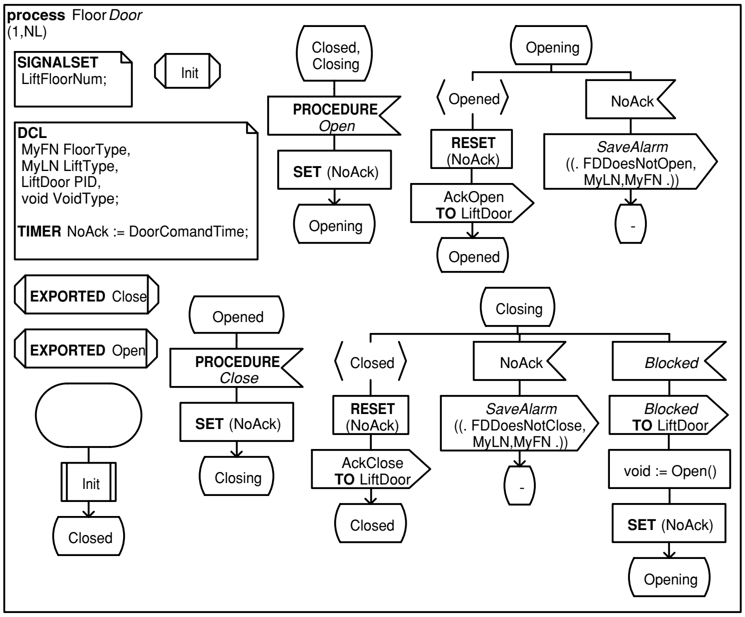


Figure 18-7. SDL Process Diagram of the `FloorDoor` Process.

interface, equipment interface, and inclusion of other design techniques (e.g., VHDL to describe hardware).

Although the SDL Abstract Data Types (ADTs) provide a way to define passive objects, they are better described by UML classes. Consequently, the detailed design ADTs are translated to UML classes and their relationships depicted in UML class diagrams, as exemplified by Fig. 18-8.

The reuse of external passive objects is facilitated by UMLs, and also SDLs, encapsulation, and inheritance mechanisms.

5. TEST DESIGN

In this phase, the communication between all the elements of the system architecture is specified by means of detailed MSCs. The detailed MSCs contain the sequences of messages exchanged between the architectural elements. They are built by refining the abstract MSC of each terminal scenario from the use case model, according to the SDL architecture model. Consequently, the test design activity can be executed parallel to the architecture design and provide requirements for the detailed design phase.

In the intermediate architecture levels, the detailed MSCs represent integration tests between the concurrent objects. The last refinement step corresponds

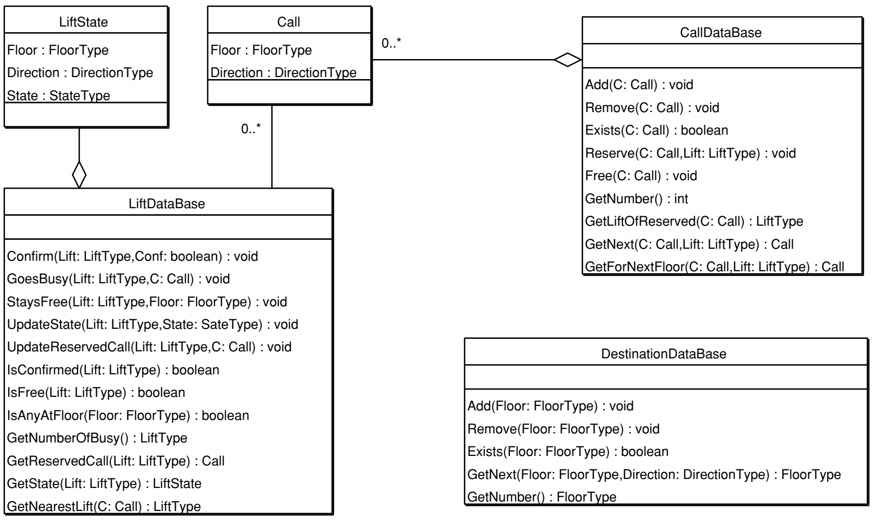

LiftState
Floor : FloorType
Direction : DirectionType
State : StateType
Call
Floor : FloorType
Direction : DirectionType
0..*
0..*
CallDataBase
Add(C: Call) : void
Remove(C: Call) : void
Exists(C: Call) : boolean
Reserve(C: Call,Lift: LiftType) : void
Free(C: Call) : void
GetNumber() : int
GetLiftOfReserved(C: Call) : LiftType
GetNext(C: Call,Lift: LiftType) : Call
GetForNextFloor(C: Call,Lift: LiftType) : Call
LiftDataBase
Confirm(Lift: LiftType,Conf: boolean) : void
GoesBusy(Lift: LiftType,C: Call) : void
StaysFree(Lift: LiftType,Floor: FloorType) : void
UpdateState(Lift: LiftType,State: SateType) : void
UpdateReservedCall(Lift: LiftType,C: Call) : void
IsConfirmed(Lift: LiftType) : boolean
IsFree(Lift: LiftType) : boolean
IsAnyAtFloor(Floor: FloorType) : boolean
GetNumberOfBusy() : LiftType
GetReservedCall(Lift: LiftType) : Call
GetState(Lift: LiftType) : LiftState
GetNearestLift(C: Call) : LiftType
DestinationDataBase
Add(Floor: FloorType) : void
Remove(Floor: FloorType) : void
Exists(Floor: FloorType) : boolean
GetNext(Floor: FloorType,Direction: DirectionType) : FloorType
GetNumber() : FloorType

Figure 18-8. Detailed Design of UML Class Diagram.

to unit tests that describe the behavior of processes (the terminal SDL architecture level). Fig. 18-9 illustrates this.

The process level MSCs can be further enriched by including in each process graphical elements with more detailed behavior, such as states, procedures, and timers. Fig. 18-10 shows the integration test corresponding to the abstract MSC of Fig. 18-4, and Fig. 18-11 represents the respective unit test for one of the internal blocks.

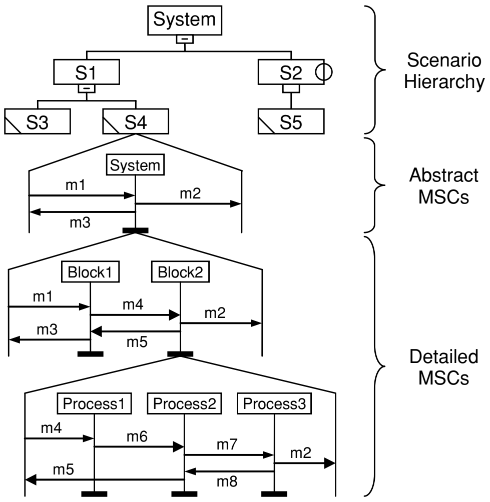


Figure 18-9. Test Design in OORT.

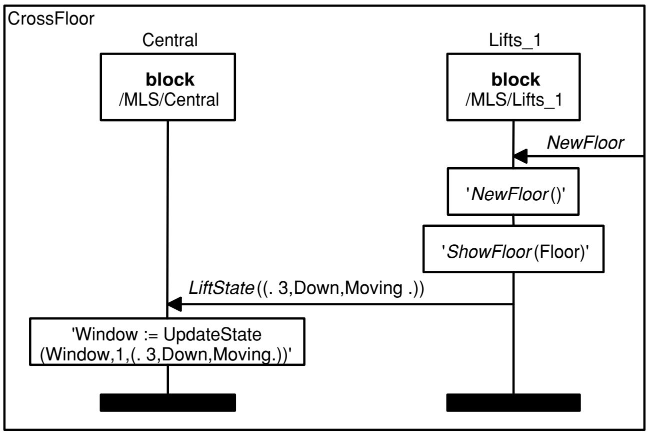


Figure 18-10. Detailed MSC with Integration Test for `CrossFloor`.

While the use case model reflects the user perspective of the system, the test design should be spread to cover aspects related to the architecture, such as performance, robustness, security, flexibility, etc.

6. SIMULATION

SDL is a formal language, and therefore permits a trustable simulation[14] of the models. The simulation of an SDL model is a sequence of steps, firing transitions from state to state.

The *Object*GEODE simulator[15] executes SDL models, comparing them with MSCs that state the expected functionalities and anticipated error situations, and it generates MSCs of the actual system behavior. It provides three operation modes: **interactive**, in which the user acts as the system environment (providing stimuli) and monitors the system's internal behavior; **random**, the simulator

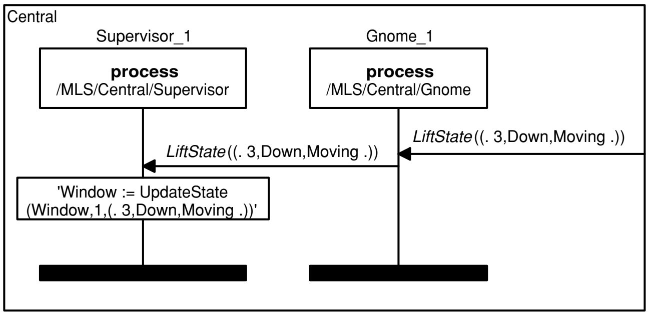


Figure 18-11. Detailed MSC with Unit Test of Block `Central` for `CrossFloor`.

picks randomly one of the transitions possible to fire; **exhaustive**, the simulator automatically explores all the possible system states.

The interactive mode can be used to do the first tests, to verify some important situations in particular. This way, the more generic system behavior can be corrected and completed. This mode is specially suited for rapid prototyping, to ensure that the system really works.

The system was very useful to detect flaws in ADTs whose operators were specified in textual SDL, as, for example, the heavy computational ADTs responsible the calls dispatching. As the simulator has transition granularity, it is not possible to go step by step through the operations executed inside one transition. The errors are detected after each transition, whenever an unexpected state is reached, or a variable has an unpredicted value. Obviously, this is not an adequate way to simulate a large number of cases.

After a certain level of confidence in the overall application behavior is achieved, it can be tested for a larger number of scenarios, in order to detect dynamical errors such as deadlocks, dead code, unexpected signals, signals without receiver, overflows, etc. This is done in the random mode, to verify if the system is being correctly built—system verification.

Although, this could be done with exhaustive simulation, it would not be efficient. The exhaustive mode requires considerable computer resources during a lot of time, and it generates a large amount of information. It is not something to be done everyday. The exhaustive simulation allows the validation of the system, i.e., to check the system against the requirements. We can verify if it implements the expected services, by detecting interactions that do not follow some defined properties, or interaction sequences that are not expected.

7. TARGETING

The *Object*GEODE automatic code generator translates the SDL specification to ANSI C code, which is independent of the target platform in which the system will run.

The SDL semantics (including the communication, process instance scheduling, time management, and shared variables) is implemented by a dynamic library which abstracts the platform from the generated code. It is also responsible for the integration with the executing environment, namely the RTOS.

In order to generate the application code, it is necessary to describe the target platform where the system will be executed. This is done by means of a mapping between the SDL architecture and the C code implementation.

The SDL architecture consists of a logical architecture of structural objects (system, blocks, processes, etc.,), in which the lower level objects (the

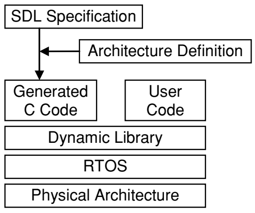


Figure 18-12. Simplified strategy for the application generation.

processes) implement the behavior of the described system. The physical implementation consists of a hierarchy of **nodes** and **tasks**. A node corresponds to one processing unit with multitasking OS, and a task is an unit of parallelism of the OS. One task can map one of the following SDL objects: system, Task/System (TS) mapping; block, Task/Block (TB) mapping; process—Task/Process (TP) mapping; process instance—Task/Instance (TI) mapping.

In the TI mapping, the complete application is managed by the target OS. In the TP mapping, the OS is in charge of the interaction between processes, whilst the management of the process instances inside the task is done by the *Object*GEODE's SDL virtual machine (VM).

In the TB mapping, the OS manages the communication between blocks, while the SDL VM executes the SDL objects inside each block. Finally, the TS mapping is the only possible option for nonmultitasking operating systems, for which the SDL VM manages all the application. For the MLS, the TP mapping was chosen.

After the automatic code generation, the code of any parts interacting or directly depending on the physical platform has to be supplied, preferably in the most suitable language. The ADT operators that do not interact with external devices can be coded algorithmically in SDL, and thus the respective C code will be generated. For each ADT operator one C function interface is automatically generated.

Fig. 18-12 illustrates the simplified application generation scheme. If some parts of the SDL model are to be implemented in hardware, Daveau[16] provides a partition and a synthesis methodology.

8. CONCLUSION

The UML, MSC, and SDL, being continuously improved to international standards, facilitates the protection of development investment. The presented work shows the validity of these languages and their combined use in the implemention of embedded systems.

It is feasible to simulate a formal language like SDL, because it is defined by a clear set of mathematical rules. The *Object*GEODE provides three simulation modes suitable for different levels of system correctness. They can be applied to make early validations and to increase the frequency of an iterative development process. This allows cost reduction by decreasing the number of missed versions, i.e., it helps the designers to get closer to the "right at first time".

The SDL application is scalable, because its logical architecture is independent of the physical architecture. The mapping between objects and hardware is defined only in the targeting phase. Furthermore, with the *Object*GEODE toolset the implementation is automatic, thus limiting the manual coding to the target dependent operations. The generated application is optimized for the target platform by means of the mapping defined by the developer. Any change in the physical architecture only requires a change in the mapping, so the system specification and its logical architecture remain the same.

The adoption of a methodology based on OO graphical languages helps the designer to organize the development tasks, and build the application as a consistent combination of parts. Object-oriented visual modeling is more flexible, easier to maintain, and favors reutilization. These advantages are emphasized when appropriate tool support exists during all the engineering process phases. In fact, it is a critical factor for success, namely for simulation and targeting.

REFERENCES

1. Verilog, *ObjectGEODE Method Guidelines*. Verilog, SA (1996).
2. Object Management Group, Unified Modeling Language Specification vl.3 (March 2000); http://www.omg.org.
3. ITU-T, Recommendation Z.120: Message Sequence Chart (October 1996); http://www.itu.int.
4. ITU-T, Recommendation Z.100: Specification and Description Language (March 1993); http://www.itu.int.
5. ITU-T, Recommendation Z.100 Appendix 1: SDL Methodology Guidelines (March 1994); http://www.itu.int.
6. ITU-T Recommendation Z.100 Addendum 1 (October 1996); http://www.itu.int.
7. UML Revision Task Force, Object Modeling with OMG UML Tutorial Series (November 2000); http://www.omg.org/technology/uml/uml_tutorial.htm.
8. ITU-T, Recommendation Z.100 Supplement 1: SDL + Methodology: Use of MSC and SDL (with ASN.1) (May 1997); http://www.itu.int.
9. E. Rudolph, P. Graubmann, J. Grabowski, Tutorial on message sequence charts. *Computer Networks and ISDN Systems* **28** (12) (1996).
10. O. Faergemand, A. Olsen, Introduction to SDL-92. *Computer Networks and ISDN Systems*, **26** (9) (1994).
11. A. Olsen, O. Faergemand, B. Moller-Pedersen, R. Reed, J.R.W. Smith, *Systems Engineering Using SDL-92*. North Holland (1994).
12. E. Yourdon, *Object-Oriented Systems Design: An Integrated Approach*. Prentice Hall (1994).

13. B.P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley (1998).
14. V. Encontre, How to use modeling to implement verifiable, scalable, and efficient real-time application programs. *Real-Time Engineering* (Fall 1997).
15. Verilog, *ObjectGEODE SDL Simulator Reference Manual*. Verilog, SA (1996).
16. J.M. Daveau, G.F. Marchioro, T. Ben-Ismail, A.A. Jerraya, Cosmos: An SDL based hardware/software codesign environment. In J.-M. Bergé, O. Levia, J. Rouillard (eds.), *Hardware/Software Co-design and Co-Verification*, Kluwer Academic Publishers (1997) pp. 59–87.

Chapter 19

OPTIMIZING COMMUNICATION ARCHITECTURES FOR PARALLEL EMBEDDED SYSTEMS

Vaclav Dvorak
Dept. of Computer Science and Engineering, University of Technology Brno, Bozetechova 2, 612 66 Brno, Czech Republik; e-mail: dvorak@dcse.fee.vutbr.cz

Abstract: The paper addresses the issue of prototyping group communications in application-specific multiprocessor systems or SoC. Group communications may have a dramatic impact on the performance and this is why performance estimation of these systems, either bus-based SMPs or message-passing networks of DSPs is undertaken using a CSP-based tool Transim. Variations in computation granularity, communication algorithms, interconnect topology, distribution of data and code to processors as well as in processor count, clock rate, link speed, bus bandwidth, cache line size and other parameters can be easily accounted for. The technique is demonstrated on parallel FFT on 2 to 8 processors.

Key words: parallel embedded systems; multiprocessor simulation; group communication; performance estimation.

1. INTRODUCTION

The design of mixed hw/sw systems for embedded applications has been an active research area in recent years. Hw/sw cosynthesis and cosimulation have been mainly restricted to a single processor and programmable arrays attached to it, which were placed incidentally on a single chip (SoC). A new kind of system, application-specific multiprocessor SoC, is emerging with frequent applications in small-scale parallel systems for high-performance control, data acquisition and analysis, image processing, wireless, networking processors, and game computers. Typically several DSPs and/or microcontrollers are interconnected with an on-chip communication network and may use an operating system.

The performance of most digital systems today is limited by their communication or interconnection, not by their logic or memory. This is why we focus ourselves on optimization of interconnection networks and communication algorithms. Interconnection networks are emerging as a universal solution to the system-level communication problems for modern digital systems and have become pervasive in their traditional application as processor-memory and processor-processor interconnection. Point-to-point interconnection networks have replaced buses in an ever widening range of applications that include on-chip interconnect, switches and routers, and I/O systems[1].

Provided that processor architecture is given and fixed, the data and code distributed to processors and the load reasonably balanced, then the only thing that remains to be optimized is an interconnection network and communication. Minimizing communication times means minimizing the main source of overhead in parallel processing. Other process interactions (and sources of overhead) like synchronization and aggregation (reduction, scan) are strongly related to communication and can be modeled as such.

In this paper we wish to concentrate on the performance estimation of various interconnection networks and communication algorithms, since performance guarantees must be complied with before anything else can be decided. We will study only application-specific multiprocessors, and modeling their performance, other difficult problems such as system validation at the functional level and at the cycle-accurate level, software and RTOS synthesis, task scheduling and allocation, overall system testing, etc., are not considered. As a suitable application for performance comparison, we have selected a parallel FFT (1024) benchmark (1024 points, one dimension), in real-time environment, with the goal of maximizing the number of such FFTs per second.

2. ARCHITECTURES OF PARALLEL EMBEDDED SYSTEMS AND CMP

The performance race between a single large processor on a chip and a single-chip multiprocessor (CMP) is not yet decided. Applications such as multimedia point to CMP with multithreaded processors[2] for the best possible performance. The choice between application-specific (systolic) architectures or processors on one hand and CMP on the other is yet more difficult. CMP architectures may also take several forms such as:

- a bus-based SMP with coherent caches with an atomic bus or a splittransaction bus;
- a SMP with a crossbar located between processors and a shared firstlevel cache which in turn connects to a shared main memory;
- a distributed memory architecture with a direct interconnection network (e.g., a hypercube) or an indirect one (the multistage interconnection network, MIN).

As the number of processors on the chip will be, at least in the near future, typically lower than ten, we do not have to worry about scalability of these architectures. Therefore the bus interconnection will not be seen as too restrictive in this context.

Some more scalable architectures such as SMP with processors and memory modules interconnected via a multistage interconnection network (the so-called "dancehall" organization) or a hw-supported distributed shared memory will not be considered as candidates for small-scale parallel embedded systems or SoCs.

Let us note that the choice of architecture can often also be dictated by a particular application to be implemented in parallel, e.g., broadcasting data to processors, if not hidden by computation, may require a broadcast bus for speed, but on the contrary, all-to-all scatter communication of intermediate results will be serialized on the bus and potentially slower than on a direct communication network. The next generation of internet routers and network processors SoC may require unconventional approaches to deliver ultrahigh performance over an optical infrastructure. Octagon topology[3] suggested recently to meet these challenges was supposed to outperform shared bus and crossbar on-chip communication architectures. However, it can be easily shown that this topology with the given routing algorithm[3] is not deadlock-free. Some conclusions like the previous one or preliminary estimation of performance or its lower bound can be supported by back-of the-envelope calculations, other evaluations are more difficult due to varying message lengths or irregular nature of communications. This is where simulation fits in.

With reference to the presented case study, we will investigate the following (on-chip) communication networks:

1. fully connected network
2. SF hypercube
3. WH hypercube
4. Multistage interconnection network MIN (Omega)
5. Atomic bus

The number of processors p = 2, 4, and 8. The problem size of a benchmark (parallel 1D-FFT) will be n = 1024 points.

3. THE SIMULATION TOOL AND DESCRIPTION LANGUAGE

Performance modeling has to take the characteristics of the machine (including operating systems, if any) and applications and predict the execution time. Generally it is much more difficult to simulate performance of an application in shared address space than in message passing, since the events of interest are not explicit in the shared variable program. In the shared address space, performance modeling is complicated by the very same properties that make developing a program easier: naming, replication and coherence are all implicit, i.e., transparent for the programmer, so it is difficult to determine how much communication occurs and when, e.g., when cache mapping conflicts are involved [4].

Sound performance evaluation methodology is essential for credible computer architecture research to evaluate hw/sw architectural ideas or trade-offs. Transaction Level Modeling (TLM)[5] has been proposed as a higher modeling abstraction level for faster simulation performance. At the TLM level, the system bus is captured as an abstract 'channel', independent of a particular bus architecture or protocol implementation. A TLM model can be used as a prototype of the system and for early functional system validation and embedded software development. However, these models do not fully exploit the potential for speedup when modeling systems for exploring on-chip communication tradeoffs and performance. On the other hand, commonly used shared-memory simulators rsim, Proteus, Tango, limes or MulSim[6], beside their sophistication, are not suitable for message passing systems.

This made us reconsider the simulation methodology for sharedmemory multiprocessors. Here we suggest using a single CSP-based simulator both for message passing as well as for shared address space. It is based on simple approximations and leaves the speed vs. accuracy tradeoff for the user, who can control the level of details and accuracy of simulation.

The CSP-based Transim tool can run simulations written in Transim language[7]. It is a subset of Occam 2 with various extensions. Transim is naturally intended for message passing in distributed memory systems. Nevertheless, it can be used also for simulation of shared memory bus-based (SMP) systems – bus transactions in SMP are modeled as communications between node processes and a central process running on an extra processor. Transim also supports shared variables used in modeling locks and barriers. Until now, only an atomic bus model has been tested; the split-transaction bus requires more housekeeping and its model will be developed in the near future.

The input file for Transim simulator tool contains descriptions of software, hardware, and mapping to one another. In software description, control statements are used in the usual way, computations (integer only) do not consume simulated time. This is why all pieces of sequential code are completed or replaced (floating point) by special timing constructs SERV ( ). Argument of SERV ( ) specifies the number of CPU cycles taken by the task. Granularity of simulation is therefore selectable from individual instructions to large pieces of code. Explicit overhead can be represented directly by WAIT( ) construct. Data-dependent computations can be simulated by SERV construct with a random number of CPU cycles. Some features of an RT distributed operating system kernel, originally supported by hw in transputers, are also built into the simulator, such as process management, process priorities (2 levels only), context switching, timers, etc.

The NODE construct in hardware description is used to specify the CPU speed, communication model and other parameters; otherwise the default values are used. The mapping between software and hardware, between processes and processors, is made through the MAP construct. Parallel processes on different processors, one process per processor, are created by PLACED PAR construct for MPMD or by replicated PLACED PAR for SPMD model of computation.

4. THE PARALLEL FFT BENCHMARK PROGRAM

We will illustrate the technique of optimization of communication architecture on the problem of computing the 1D-, n-point-, discrete Fourier transform on p processors in O((n log n)/p) time. Let p divide n, $n = 2^q$ is a power of two and $n \geq p^2$. Let the n-dimensional vector x [n$\times$1] be represented by matrix X [n/p $\times$ p] in row-major order (one column per processor). The DFT of the vector x is given by

$$Y = W_n X = W_P \left[S^* W_{n/p} X\right]^T, \tag{1}$$

where S [n/p $\times$ p] is the scaling matrix, * is elementwise multiplication and the resulting vector y [n $\times$ 1] $= W_n x$ is represented by matrix Y [n/p $\times$ p] in column major order form (n/p^2 rows per processor). Operation denoted by T is a generalized matrix transpose that corresponds to the usual notion of matrix transpose in case of square matrices[8].

The algorithm can be performed in the following three stages. The first stage involves a local computation of a DFT of size n/p in each processor, followed by the twiddle-factor scaling (element-wise multiplication by S). The second stage is a communication step that involves a matrix transposition. Finally, n/p^2 local FFTs, each of size p, are sufficient to complete the overall FFT computations on n points. The amount of computation work for the sequential FFT of an n-element real vector is $(n/2)\log_2 n$ "butterfly" operations, where one butterfly represents 4 floating point multiplications and 6 additions/subtractions (20 CPU clocks in simulation). In parallel implementation the computation work done by p processors is distributed at stages 1 and 3, but the total amount of work in terms of executed butterfly operations is the same,

$$p\left[\frac{n}{2p}\log\frac{n}{p} + \frac{n}{p^2}\frac{p}{2}\log p\right] = \frac{n}{2}\log n. \tag{2}$$

Let us note that the work done in stage 1 proportional to ($\log n - \log p$) is much larger than the work done in stage 3, proportional to $\log p$. The only overhead in parallel implementation is due to a matrix transposition. The matrix

Table 19-1. The lower bound on total exchange (AAS) communication times

Topology	Bisection width b	Bisection band-width Gbit/s	Time of AAS [μs], the lower bound
Full connection	$p^2/2 = 32$	3.2	3.2
MIN Omega	4 log p = 12	1.2	8.53
Bus	–	1.0	35.8
SF cube	p = 8	0.8	12.8
WH cube	p = 8	0.8	12.8

transposition problem is equivalent to all-to-all scatter (AAS) group communication. Clearly, it requires 1 step in a fully connected topology, p/2 steps (a lower bound) in the SF-hypercube, p-1 steps in the WH-hypercube or a MIN, and finally p(p-1) bus transactions on a bus. The lower bound on the AAS communication time can be obtained using a network bisection bandwidth and the required amount of data movement across bisection (under the assumption that data movement within each part may be overlapped with data movement between both parts). Bisection partitions a network into two parts, so that in AAS communication each node in one part has to communicate with all nodes in the other part using b channels cut by bisection. We therefore have (p/2) $\times$ (p/2) messages of size n/p^2 real numbers, i.e., (n/4) $\times$ 4 byte or n bytes. A different number p(p-1) of shared memory communications (read miss bus transactions) are needed for the SMP. Lower bounds of communication times are summarized in the Table 19-1 using parameters from the next subsection 5 and assuming 10 bits/byte.

FFT processing will be done continuously in real time. Therefore loading of the next input vector from outside and writing the previous results from processors to environment will be carried out in the background, in parallel with three stages of processing of the current input vector (with the first stage of processing only in the shared memory case). Since computing nodes are identical in all architectures, only the duration of "visible" AAS communication makes a difference to the performance. Communication time overlapped by useful processing is invisible (does not represent an overhead) and ideally all communications should be hidden in this way.

5. PARAMETERS OF SIMULATED ARCHITECTURES AND RESULTS OF SIMULATION

Six architectures simulated in the case study are listed in Table 19-2 together with the execution times. The CPU clock rate is 200 MHz in all 6 cases, the

Table 19-2. Parallel FFT execution times in μs for six analyzed architectures

Topology	p = 2	p = 4	p = 8
Full connection	436.8	180.8	138
MIN—COSP		230.4	173.1
Bus—SMP	363.5	304.7	321.6
SF cube		272	182.8
WH cube		230	174.4

external channel speed of 100 Mbit/s (12 MB/s) is used for serial links in all message-passing architectures, whereas bus transfer rate for SMP is 100 MB/s. Downloading and uploading of input data and results were supposed to continue in the background in all processors simultaneously at a 8-times higher rate than the link speed, which is almost equivalent to the bus speed in SMP case. In message-passing architectures the AAS communication was overlapped with submatrix transposition as much as possible. Optimum routing algorithm for SF hypercube and AAS communication requires p/2 steps and uses schedule tables shown in Fig. 19-1. For example two nodes with mutually reversed address bits (the relative address RA = 7) will exchange messages in step 2, 3, and 4 and the path will start in dimension 1, then continue in dimension 0 and finally end in dimension 2. In case of WH hypercube, dimension-ordered routing is used in every step i, i = 1, 2, . . . , p-1, in which src-node and dst-node with the relative addresses RA = src ⊕ dst = i exchange messages without any conflict-over disjoint paths.

The small cluster of (digital signal) processors, referred to as COSP in Table 19-2, uses a centralized router switch (MIN of Omega type) with sw/hw overhead of 5 μs, the same as a start-up cost of serial links, and WH routing. The algorithm for AAS uses a sequence of cyclic permutations, e.g., (01234567), (0246)(1357), . . . , (07654321) for p = 8. All these permutations are blocking and require up to log p = 3 passes through the MIN.

Finally a bus-based shared memory system with coherent caches (SMP) has had 100 MB/s bus bandwidth, 50 MHz bus clock, and the miss penalty of 20 CPU clocks. We will assume an atomic bus for simplicity and a fair bus

step	RA in dimension	
	0	1
1	3	2
2	1	3

step	relative addr. used in dimension		
	0	1	2
1	3	6	4
2	1	7	6
3	7	2	5
4	5	3	7

Figure 19-1. Optimum schedule for AAS in all-port full-duplex 2D- and 3D SF hypercubes.

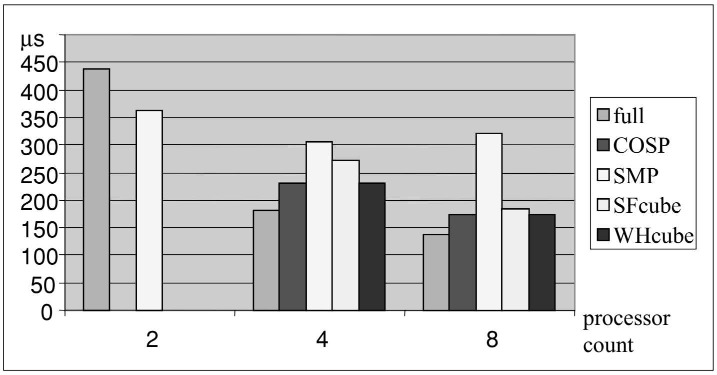


Figure 19-2. Comparison of execution times [ms] for six architectures.

arbitration policy. Other types of bus arbitration (priority-based, random, etc.,) may also be simulated. The cache block size is 16 bytes and the size of the cache is assumed to be sufficient to hold input data (a real vector), intermediate data after the first stage of FFT (a complex vector) as well as the results (a complex vector). In the worst case ($p = 2$) the size of all these vectors will be around 10 kB, if we use REAL32 format. We assume I/O connected via a bus adapter directly to the cache. To avoid arbitration between CPU and I/O, the next input and previous results are transferred in/out during the first stage of the FFT algorithm.

The results summarized in Table 19-2 and plotted in Fig. 19-2 deserve some comments. A fully connected network of processors is the fastest architecture for 8 processors, but the slowest for 2 processors. The reason is that communication is mostly seen as an overhead, but gets better overlapped with communication when p increases. The cluster of DSPs (COSP row in Table 19-2) starts with $p = 4$ and increasing the number of processors from 4 to 8 does not make much sense because it has a small influence on speed.

In the SMP with shared bus, processors write the results of the n/p-point FFT computed in stage 1 into the local caches and do the transposition at the same time. This means that consecutive values of FFT will be stored with a stride required by the rule of matrix transposition. The following read requests by other processors at the beginning of stage 3 will generate read misses: at cache block size 16 bytes, one miss always after 3 hits in a sequence. Fresh cache blocks will be loaded into requestor's cache and simultaneously into the shared memory. A prefetch of cache blocks has been simulated without an observable improvement in speed, most probably due to bus saturation. This is even worse for 8 processors than for 4, see Fig. 19-2.

As for hypercubes, the WF hypercube is superior and gives the same results as a cluster of DSPs. Slightly worse performance than that of fully connected processors is balanced by much simpler interconnection and by a lower number of communication ports.

6. CONCLUSIONS

The performance study of the parallel FFT benchmark on a number of architectures using Transim tool proved to be a useful exercise. Even though the results of simulations have not been confronted with real computations, they can certainly serve to indicate serious candidate architectures that satisfy certain performance requirements. The approximations hidden in the simulation limit the accuracy of real-time performance prediction, but the level of detail in simulation is given by the user, by how much time they are willing to spend on building the model of hw and sw. For example, modeling the split-transaction bus or the contention in interconnection network for WH routing could be quite difficult. The latter was not attempted in this case study since the FFT benchmark requires only regular contention-free communication. This, of course, generally will not be the case. Nevertheless, simulation enables fast varying of sw/hw configuration parameters and studying of the impact of such changes on performance, free from the second-order effects. In this context, the CSP-based Transim simulator and language proved to be very flexible, robust, and easy to use. Future work will continue to include other benchmarks and analyze the accuracy of performance prediction.

ACKNOWLEDGMENTS

This research has been carried out under the financial support of the Research intention no. CEZ: J22/98: 262200012—"Research in information and control systems" (Ministry of Education, CZ) and the research grant GA 102/02/0503 "Parallel system performance prediction and tuning" (Grant Agency of Czech Republic).

REFERENCES

1. J. Duato, S. Yalamanchili, L. Ni, *Interconnection Networks—An Engineering Approach.* Morgan Kaufman Publishers (2003).
2. J. Silc, B. Robic, T. Ungerer, *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer-Verlag (1999), ISBN 3-540-64798-8.
3. F. Karim, A. Nguyen, *An Interconnect Architecture for Networking Systems on Chips. IEEE Micro*, pp. 36–45, Sept.–Oct. 2002.

4. D.E. Culler et al., *Parallel Computer Architecture*. Morgan Kaufmann Publ., p. 189 (1999).
5. T. Grötker, S. Liao, G. Martin, S. Swan, *System Design with SystemC*. Kluwer Academic Publishers (2002).
6. http://heather.cs.ucdavis.edu/∼matloff/mulsim.html
7. E. Hart, *TRANSIM–Prototyping Parallel Algorithms*, (2nd edition). University of Westminster Press London, (1994).
8. A. Zomaya, *Parallel and Distributed Computing Handbook*. McGraw Hill, p. 344 (1996).

Chapter 20

REMARKS ON PARALLEL BIT-BYTE CPU STRUCTURES OF THE PROGRAMMABLE LOGIC CONTROLLER

Mirosław Chmiel and Edward Hrynkiewicz
Institute of Electronics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland; e-mail: Miroslaw.Chmiel@polsl.pl, eh@boss.iele.polsl.gliwice.pl

Abstract: The paper presents some hardware solutions for the bit-byte CPU of a PLC, which are oriented for maximum optimisation of data exchange between the CPU processors. The optimization intends maximum utilization of the possibilities given by the two-processor architecture of the CPUs. The key point is preserving high speed of instruction processing by the bit-processor, and high functionality of the byte-processor. The optimal structure should enable the processors to work in parallel as much as possible, and minimize the situation, when one processor has to wait for the other.

Key words: PLC; CPU; Bit-Byte structure of CPU; control program; scan time.

1. INTRODUCTION

One of the main parameters (features) of Programmable Logic Controllers (PLC) is execution time of one thousand control commands (scan time). This parameter evaluates the quality of PLC. Consequently, designing and construction are important tasks of the CPU which should have a structure enabling fast control program execution. The most developed CPUs of PLCs of many well-known manufacturers are constructed as multiprocessor units. Particular processors in such units execute the tasks assigned to them. In this way we obtain a unit, which makes possible parallel operation of several processors. For such a CPU the main problem to be solved is the method of task-assuming in particular processors and finding a structure of CPU capable of realisation of such task-assigning in practice as shown by Michel (1990).

The bit-byte structure of CPU in which task assignment is predefined is often met in real solutions. The tasks operating on discrete input/outputs are executed by the bit-processor. Such processors may be implemented in programmable structures as PLD or FPGA and has been demonstrated (Chmiel et al., 1995b; Hrynkiewicz, 1997). It brings the positive effects in user program execution time (fast operating processor). On the other hand a byte-processor (word-processor) is built on the base of a standard microprocessor or embedded microcontroller. The byte-processors are used for the control of analogue objects, for numeric data processing, and for the execution of the operations indirectly connected to user (control) program but connected to the operating system of the programmable controller of a CPU. A set of such operations consists of timer servicing, reading-out of the input states, setting of the outputs, LAN servicing, communication to the personal computer, and so on.

A very interesting problem, though a difficult task, is the realisation of programmable controller timer module as shown in (Chmiel et al., 1995a). A time interval is counting, asynchronously, to the program loop execution. It causes difficulties with testing of an end of a counted interval. At long time of program loop execution and short counted time intervals a large error may occur. The accuracy of time intervals counting may be increased by special program tricks, but achieving good results is typically connected with the prolongation of the control program loop. In some programmable controllers the end of time interval counting interrupts the control program and the service procedure for this interrupt is invoked. However, the number of interrupts is typically limited and only a few timers can act in this way. This is why it would be worthwhile to reflect on a way of improving an accuracy of time counting in the programmable controllers. The other matter is connected with this problem. As mentioned earlier operation speed is one of the most important parameters of programmable controllers. Typically, operation speed is closely connected to the scan time. However, it seems that throughput time more precisely describes the dynamic features of a programmable controller. Naturally it may be said that throughput time is closely linked to the scan time unless a programmable controller does not execute a control program in a serial-cyclic way. Let us imagine that a programmable controller operates on the rule based on processing of the segments (tasks) of the control program. These segments are triggered only by the changes of the input signals (input conditions). In this situation one can tell the throughput time (response time) but it would be difficult to tell the time of program loop execution. It would be possible only to define the mean time of program loop execution for a given application. For the application where the signal changes sparsely. The mean time of program loop execution will be much less than the maximum time evaluated for the execution of a whole program. In particular applications the certain group of signals may change more often than the other signals. The segments of the control program triggered by these signals will be executed more often than the other segments. To avoid a situation where two or more tasks are triggered at the same moment it would be necessary to assign the priorities to the control program segments. The described method of programmable controller operation changes the approach to preparing a control program but it seems to the authors that in such a programmable controller the problems with, for example, timers will be easier. It is not necessary to observe the moment when time interval will be completed. At the end of the time interval, counting the suitable segment may be called and executed. It means that the currently executed program segment should be interrupted and this depends on the priorities assigned to the particular program segments.

Such type of programmable controller CPUs will be the subject of future work, while in this paper the few proposals of programmable controller bit-byte CPU structures are presented. These are CPUs with serial-cyclic program execution but they are structurally prepared for event-triggered operation.

2. THE REQUIREMENTS FOR PROGRAMMABLE CONTROLLER CPUs

The aim of the work results as described in the paper were design and implementation of programmable controller CPU based on bit-byte structure. The main design condition was maximum speed of control program execution. This condition should be met rather by elaborating suitable structure rather than application of the fastest microprocessors. Additionally, it was assumed that bit-processor will be implemented using catalogue logic devices or programmable structures whereas the byte-processor will be used by the microcontroller 80C320 from Dallas Semiconductor. The CPU should be capable of carrying out the logical and arithmetical operations, conditional and unconditional jumps, sensing the input states, and setting or resetting outputs, timers, counters, and so on.

In the simplest case, each programmable control device might be realised as a microprocessor device. We have to remember about applications in which we are going to use the constructed logic controller. These applications force special requirements and constraints. Controlled objects have a large number of binary inputs and outputs, while standard microprocessor (or microcontroller) operates mainly on bytes. A instruction list of those devices is optimised for operation on byte or word variables (some of them can carry out complicated arithmetical calculation) that are not required in industrial applications. Each task is connected with reading external data, computation, and writing computed data to the outputs. Logical instructions like AND or OR on individual bits take the same amount of time. The number of binary inputs and outputs in greater units reach number of thousands. In such cases parallel computation of all inputs and outputs is impossible. In this situation all inputs and outputs must be scanned and updated sequentially as fast as possible. If we wish to achieve good control parameters, the bits operation should be done very quickly.

The creation of a specialised bit-processor, which can carry out bit operation very quickly, is quite reasonable. If there is a need to process byte data, for example from AD converters or external timers, the use of additional 8, 16, or even 32, bits processor or microcontroller is required. The General structure of the device is presented by Getko (1983).

The solution consists of two processors. Each of them has its own instruction set. An instruction decoder recognises for which processor an instruction was fetched and sends activation signals to it.

The basic parameter under consideration was program execution speed. Program execution speed is mainly limited by access latency of both processors to the internal (e.g., counter timers) and external (e.g., inputs and outputs) process variables. The program memory and the instruction fetch circuitry also influence system performance. In order to support conflictless cooperation of both processors and maintain their concurrent operations the following assumption were made:

- both processors have separate data memory but process image memory is shared between them. It seems a better solution when each processor has independent bus to in/out signals that are disjoint sets for both processors. In order to remove conflicts in access to common resources (process memory) two independent bus channels are implemented to process memory, one for each processor. This solution increases cost but simplifies processor cooperation protocol, especially by the elimination of the arbitration process during access to common resources;
- presented circuit has only one program memory. In this memory there are stored instructions for bit and byte processors. Byte processor usually executes a subprogram which has a set of input parameters. The byte processor would highly reduce instruction transfer performance by accessing program memory in order to fetch an invariant part of subroutine that is currently executed. Donandt (1989) proposed a solution that implements separated program memory for byte processors. In the presented case we decided to implement two program memories for byte processor. In common program memory, subprogram calls are stored with appropriate parameter sets. In local program memory of byte processor there are stored bodies of subroutines that can be called from controller program memory. It allows saving program memory by replacing subprograms of byte processor with subprogram calls. Subprograms implement specific instructions of the PLC, which are not typical for general purpose byte processors;
- in order to reduce access time to in/out signal, process memory was replaced by registers that are located in modules. Content refresh cycle of

these registers will be executed after the completion of the calculation loop. In some cases register update can be executed on programmer demand. Presented solution gives fast access to in/out modules with relatively low requirements for a hardware part. It is also possible to bypass registers and directly access module signals.

The bit processor that operates as a master unit in the CPU allows speeding up operation of the controller. There are also disadvantages connected with this architecture, which can be easily compensated. Main limitation is microprogrammable architecture of bit processor that has limited abilities in comparison to standard microprocessor or microcontroller. Proper cooperation protocol of both processors allows eliminating of those limitations. Separated bus channels as well as separated program and data memories assure concurrent operation of the bit and byte processors without requirement of arbitration process.

The structure of the designed controller must be as simple and cheap as possible. There must be minimal influence on the execution speed of byte as well bit processor (all processors should be able to operate with highest possible throughput). It is obvious that not all assumptions can be fully satisfied.

Following assumptions were made in order to support two processors in concurrent operations:

- separate address buses for bit and byte processors;
- two data buses: for bit-processor and for microcontroller;
- two controls bus, separate for microcontroller and bit-processor.

3. SELECTED STRUCTURES OF THE BIT-BYTE CENTRAL PROCESSING UNIT

In this section the presented concept is of bit and byte processors cooperation that allows achieving maximal execution speed by logic controller.

Two components were the base for research and design works. These components are program memory (memories) and instruction fetch by both types of processors. A satisfying solution must reduce the number of accesses to common memory. The access cycle to common memory must also be as fast as possible. Better-tailored subprograms for byte processors allow reducing of a number of instructions for it in a whole instruction stream.

Data exchange protocol between processors and access to common resources (timers, counters) are other problems that must be addressed.

The next problem is the method of information exchange between the processors and an access to the timers and counters. Timers and counters are related to both processors because they are implemented in a software way in the byte-processor, while their state is more often used by the bit-processor.

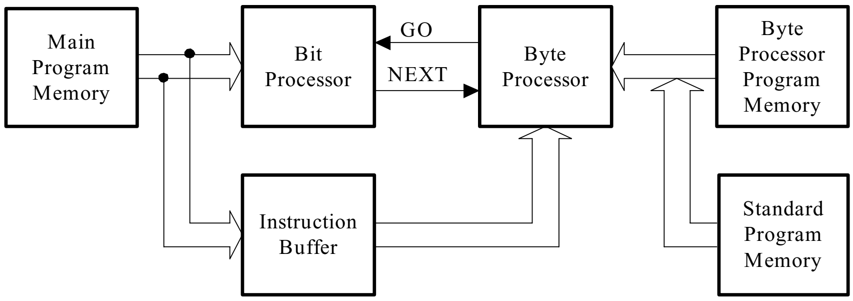


Figure 20-1. An illustration of the commands transfer to the CPU processors.

The idea of command fetch and passing it to the appropriate processor of the PLC central unit is presented in Fig. 20-1. There is one common program memory called main program memory and two auxiliary memories that are used by the byte processor. One of the processors fetches instructions. Next the instruction is decoded. If the currently active processor can execute instructions it is passed on to the execution stage. In case it cannot be executed the NEXT signal is asserted. It keeps the processors synchronized to each other. Bit processor operates faster then byte processor and usually more instructions are addressed to the bit processor then to the byte processor. In that case the bit processor should be a master processor, which fetches instruction and eventually passes it to the byte processor. Operations executed by the byte processor are represented by subprogram calls that represent basic tasks for byte operation like timers/counters updates or communication tasks. The byte-processor is a kind of coprocessor that processes requests from the bit processor.

The bit processor fetches commands from the main program memory pointed by the program counter. The program counter is implemented inside the bit processor. Next instruction is decoded and, when it is dedicated for the bit processor, signal NEXT is set to 0 and the bit processor can execute instructions. In case instructions must be passed to the byte processor NEXT signal is set to 1 and the instruction is passed to the buffer register. After passing instructions to the byte processor the bit processor is waiting for activity of GO signal that allows it to resume operation.

As assumed, both processors can operate almost independent of each other. They are able to execute instructions from respective memories and access in/out modules simultaneously. One problem that must be solved is data exchange between processors.

There has to be a common memory through which processors are able to exchange data for the following purposes:

- setting and clearing flags that request execution of specific tasks instead of exchanging whole instructions.
- data transfers.

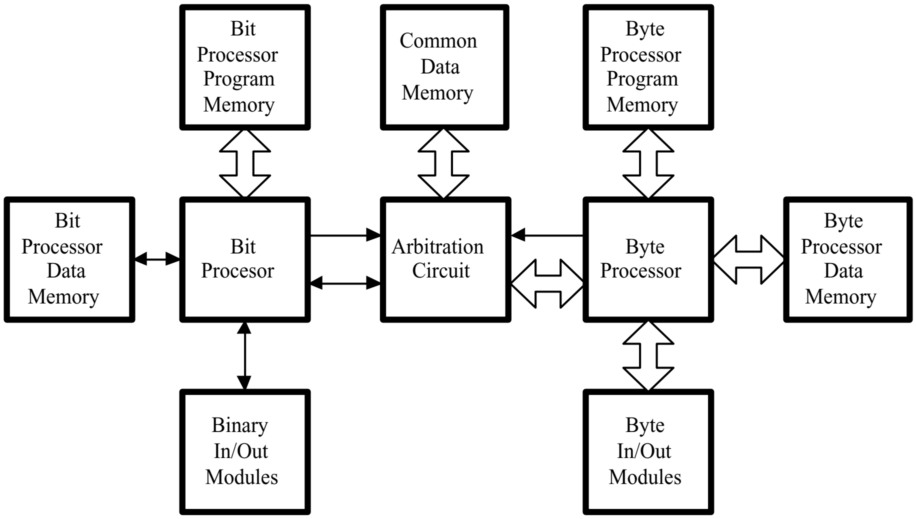

Bit Processor Program Memory
Common Data Memory
Byte Processor Program Memory
Bit Processor Data Memory
Bit Procesor
Arbitration Circuit
Byte Processor
Byte Processor Data Memory
Binary In/Out Modules
Byte In/Out Modules

Figure 20-2. Block diagram of the two processors CPU with common memory.

This conception is presented in Fig. 20-2. This figure concentrates on data exchange. It is based on a similar idea as presented earlier. This solution assumes common program memory for both processors. Each of them has unique operation codes. One of the processors fetches operation code and recognises it. If fetched instruction is assigned to it, it is immediately executed in other cases and is sent to the second processor for execution.

The unit is equipped with three memory banks for the control program:

- program memory for bit processor;
- program memory for byte-processor includes standard procedure memory; Such CPU has three states of operation:
- both processors execute control program simultaneously;
- one processor operates;
- bit-processor executes control program while byte-processor e.g., actualises the timers.

The modification of the above solution, referring to the first conception is the unit where the bit-processor generates pulses activating the sequential tasks in the byte-processor. These tasks are stored in suitable areas of the byte-processor memory.

Finally the CPU structure presented in the Fig. 20-2 was accepted. This structure was additionally equipped with the system of fast data exchange keeping PLC programming easy. This system—in simple words—ensures that the processors do not wait to finish their operations but they execute the next commands up to the moment when command of waiting for result of

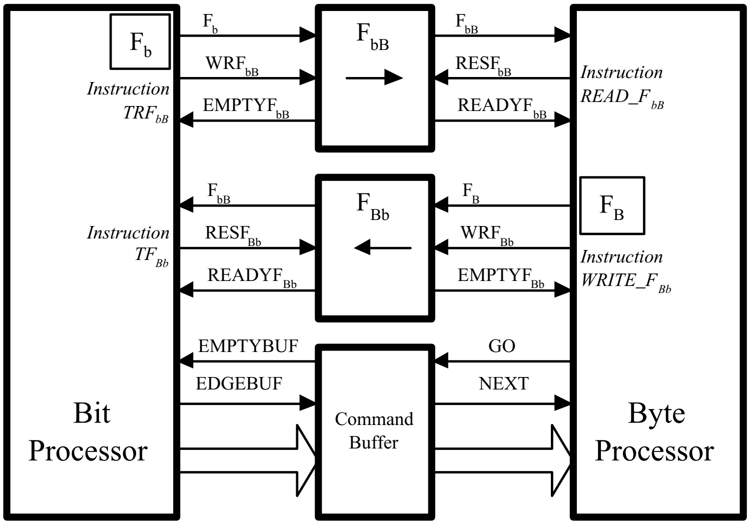


Figure 20-3. The details of synchronisation of CPU processors.

operation carried-out by the second processor occurs. The important point is the suitable program compiler and the way the control program is written by the designer.

The bit-processor delivers commands to the byte-processor through the command buffer informing it by means of NEXT = 0–signal. On the other hand the byte-processor, after accepting a command, sends it to the bitprocessor as confirmation by asserting EMPTYBUF signal.

The processors can exchange the result of recently executed operations through F_{bB} and F_{Bb} flip-flops. Processors must be able to read and write appropriate registers in order to pass on information. This justifies the use of special instructions that are marked in Fig. 20-3. by Chmiel and Hrynkiewicz (1999). From the side of bit processors those are two transfer instructions: TRF_{bB} that allows writing state of condition register to F_{bB} and TF_{Bb} that allows testing state of F_{Bb} flip-flop. A similar set of two instructions is implemented for the byte-processor. There are $READ\_F_{Bb}$ that reads contents of F_{Bb} and $WRITE\_F_{Bb}$ that transfers content of internal condition flip-flop to F_{Bb}.

The two following situations can cause one of the processors to wait for another reducing speed of program execution:

- the first processor has not yet executed operation expected by the second processor and this one has to wait for the result ($READYF_{Bb} = 0$ or $READYF_{bB} = 0$);
- the second processor has not yet received the previous result and the first one cannot write the next result ($EMPTYF_{bB} = 0$ or $EMPTYF_{Bb} = 0$).

To reduce possibility of occurrence of wait condition programs should be written and compiled in such a way as to get these two processors working in parallel as far as possible. However, in the second case one can take into account the solution based on the increased number of the accessible data exchange flip-flops or on assignment of common memory area for the data exchange purpose.

At that time it appears that there is the need to assign flags to every task. One can try to solve the flag problem in the following ways:

- the fixed flag can be assigned to every type of operation (this solution is not flexible, both processors have to access to the common memory area frequently, or many condition flip-flops have to be used);
- the successive tasks will use successive flags and this process will repeat itself periodically after the number of flags runs out. The assignment process can be led automatically by the compiler. However this solution can be applied for the instruction sequences not disturbed by program jumps (except for the jump to the beginning of the program loop);
- the third way is to charge the program designer with the duty of flag assignment. In this case flags are passed instead of markers. Results of program execution in one of the processors is passed through a special memory area whose functionality is similar to markers.

As presented earlier 2nd and 3rd solutions can be implemented in hardware. Number of flip-flops must be large enough to transfer all possible conditions in the longest program. In general the number of flip-flops used as markers is proportional to the length of the program, which is determined by the capacity of its program memory.

Condition flip-flops are grouped into two sets that pass information in both directions between the two processors. The simplest implementation writes results of the operation to the queue. The opposite processor reads results from the queue as needed. The flag system is implemented as a FIFO register that allows storing of all markers in order of their appearence. Processor writes condition flag to the flip-flop register pointed by condition counter. Opposite processor reads condition flags and selects current register by its condition counter. In this way the circular buffer was designed which allows for reading and writing from different registers. Size of the circular buffer must be large enough to store all required information passed among the processors.

Presented solutions offer extremely fast operation requiring only one clock cycle from the side of each processor. Unfortunately this solution is expensive in comparison to common memory, which can also be used as memory for timers, counters, and flags. The memory accessed by two processors requires a special construction or arbitration system. In order to avoid the arbitration process in an access cycle, two special gate memory must be used that allow simultaneous access to memory array by two processes.

4. SYNCHRONISATION OF THE PROCESSORS

The bit-byte CPU (Fig. 20-2) can work in one of two modes:

- dependent operation mode—the parallel–serial work of processors with exchange of the necessary data, coordinated by the bit-processor, which is faster. It is the basic work-mode of the designed CPU. This mode uses all the conceptions presented in Fig. 20-1;
- independent operation mode—fully parallel. Both units work fully independent, each one has its own program so time is not wasted for transferring of the commands. There is also no data exchange between the processors. Unfortunately such a mode is applicable only for some control programs.

Dependent operation mode is basic operation mode. It employs serial-parallel operation with essential information exchange among two processors. The faster processing unit coordinates operation. Detailed operation of the PLC can be described as follows:

- when conditional instruction is encountered, data is passed from the bit processor to the byte processor. In order to pass condition value the bit processor must execute TRF_{bB} instruction. Execution of this instruction is postponed until line $EMPTYF_{bB}$ is asserted. This invalidates the state of the F_{bB} register and allows writing of new information in it. Writing F_{bB} flipflop completes data transfer from the bit processor to the byte processor and allows fetching new instruction by the bit processor. Conditional instruction executed in the byte processor at the beginning checks the state of the F_{bB} register. This operation executes the procedure called $READ\_F_{bB}$. When line $READYF_{bB}$ is set, the microprocessor can read data from F_{bB}. After reading the condition the flip-flop line $RESF_{bB}$ is pulsed and asserts signal $EMPTYF_{bB}$ and allows transfer of the next condition from the bit processor. Finally the byte processor can start execution of the requested subprogram according to the reading of the condition flag;
- there are situations in program flow that require information from the byte processor. Usually it is a result of subprogram execution that is needed in further calculations of the bit processor. In order to fetch a condition result from the byte processor the bit processor executes TF_{Bb} instruction. Before reading F_{Bb} flip-flop state of the $READYF_{Bb}$ line is checked. When the register is written a new value line is activated ($READYF_{Bb} = 1$); in the opposite case the bit processor waits until conditional result is written to flip-flop and the line is asserted. Now data can be transferred from F_{Bb} register to internal condition register of the bit processor. At the same time the line $RESF_{Bb}$ is activated that set $EMPTYF_{Bb}$ signal to 1. This signal allows the byte processor to write a new condition result to F_{Bb} register. From the side of byte the processor subprogram $WRITE\_F_{Bb}$ checks the

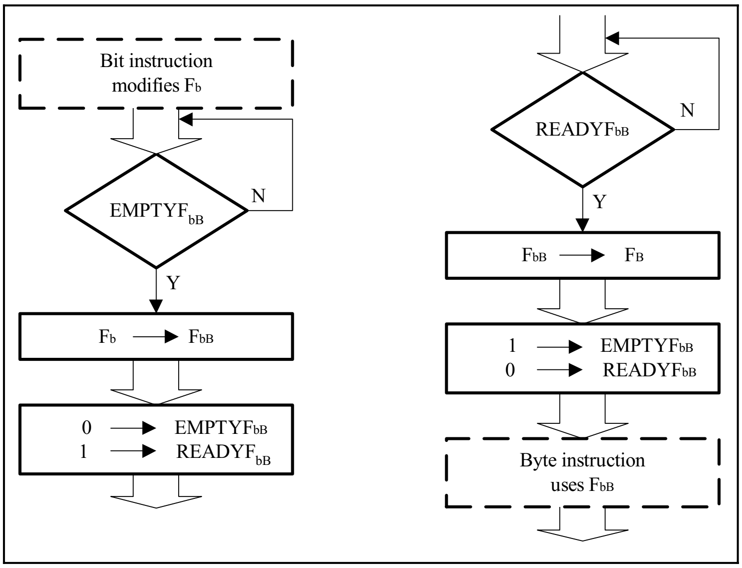


Figure 20-4. Inter-processors condition passing algorithms.

state of EMPTYF$_{Bb}$ line. Until this line is not set new condition result must not be written to F$_{Bb}$ register as it still contain valid data that should be received by the bit processor.

Such exchange of the conditional flags does not require postponing of the program execution. Proper data transfer is maintained by handshake registers that controls data flow among processors and also synchronizes program execution. When condition result is passed from one processor to another must be always executed by a pair of instructions. When data is passed from the bit processor to the byte processor those are TF$_{bB}$ and READ_F$_{bB}$. Transfer in opposite direction requires execution of instructions WRITE_F$_{Bb}$ and TRF$_{Bb}$. In Fig. 20-4 inter-processors condition passing algorithms are presented.

5. CONCLUSION

Studies on the data exchange optimization between the processors of the bit-byte CPU of the PLC have shown the great capabilities and the possible applications of this architecture.

As can be seen from the given considerations, the proposed PLC structure—or, to be more precise, organization of information exchange between both processors of a PLC central unit, allows for execution of the control programs consisting of bit command and/or word commands.

Two modes of CPU operations were considered. Basic mode—called dependent operation mode—brings worst timings than the independent operation mode. It is obvious, taking into account that both processors wait for the results of executed operation by each other. The authors thought that it is possible to work both CPU processors with exchanging of the condition flags. But this problem will be the subject of future work.

REFERENCES

M. Chmiel, E. Hrynkiewicz, Parallel bit-byte CPU structures of programmable logic controllers. In: *International Workshop ECMS*, Liberec, Czech Republic, pp. 67–71 (1999).

M. Chmiel, W. Ciążyński, A. Nowara, Timers and counters applied in PLCs. In: *International Conference PDS*, Gliwice, Poland, pp. 165–172 (1995a).

M. Chmiel, L. Drewniok, E. Hrynkiewicz, Single board PLC based on PLDs. In: International Conference *PDS*, Gliwice, Poland, pp. 173–180 (1995b).

J. Donandt, *Improving response time of Programmable Logic Controllers by use of a Boolean Coprocessor*, IEEE Comput. Soc. Press., Washington, DC, USA, 4, pp. 167–169 (1989).

Z. Getko, *Programmable Systems of Binary Control*, Elektronizacja, WKiL, Warsaw (in Polish), 18:5–13 (1983).

E. Hrynkiewicz, Based on PLDs programmable logic controller with remote I/O groups. In: *International Workshop ECMS*, Toulouse, France, pp. 41–48

G. Michel, *Programmable Logic Controllers, Architecture and Applications*. John Wiley & Sons, West Sussex, England (1990).

Chapter 21

FPGA IMPLEMENTATION OF POSITIONAL FILTERS

Dariusz Caban
Institute of Engineering Cybernetics, Wroclaw University of Technology, Janiszewskiego 11-17, 50-370 Wroclaw, Poland; e-mail: darek@ict.pwr.wroc.pl

Abstract: The paper reports on some experiments with implementing positional digital image filters using field programmable devices. It demonstrates that a single field programmable device may be used to build such a filter. By using extensive pipelining in the design, the filter can achieve performance of 50 million pixels per second (using Xilinx XC4000E devices) and over 120 MHz (in case of Spartan-3 devices). These results were obtained using automatic synthesis from VHDL descriptions, avoiding any direct manipulation in the design.

Key words: Positional filter; median filter; synthesis; FPGA implementation.

1. INTRODUCTION

The paper reports on the implementation of a class of filters used in image processing. The filtering is realised on a running, fixed size window of pixel values. Positional filtering is obtained by arranging the values in an ordered sequence (according to their magnitude) and choosing one that is at a certain position (first, middle, last, or any other). Thus, the class of filters encompasses median, max, and min filtering, depending on the choice of this position.

There are various algorithms used in positional filtering[1,2]. These are roughly classified into three groups: compare-and-multiplex[3], threshold decomposition[4], and bit-wise elimination[5,6,7]. All these can be used with the currently available, powerful FPGA devices. However, the bit-wise elimination method seems most appropriate for the cell array organisation.

Some specific positional filters have commercial VLSI implementations. There is no device that can be configured to realise any position filtering. Even if only median, min, or max filtering is required, it may be advantageous to use FPGA devices, as they offer greater versatility and ease of reengineering. Of course, FPGA implementations are particularly well suited for application in experimental image processing systems.

The first attempts to use FPGAs as reconfigurable image filters were reported almost as soon as the devices became available[8,9,10]. The devices proved to be too inefficient for full-fledged use, forcing the designers to limit the window size, pixel rates, or the width of their bit representations. This is no longer the case since the Virtex family of devices became available[11].

Filter reconfiguration can be fully utilized only if there is an easy route to obtain new configuration variants. In case of FPGA implementation, this is offered by autosynthesis: new algorithms are described in terms of a hardware description language and the rest is done by the design tools with no human interaction. The results in the paper were obtained using the Xilinx Foundation 4.1i tools with FPGA Express (XC4000E and Virtex-2 devices) or the Xilinx ISE 6.2i with built-in XST tool.

2. BIT-WISE ELIMINATION METHOD

Positional filtering is based on reordering of the pixel values according to their magnitude. Let us denote the kth value in the reordered sequence by P_n^k (where n is the length of the sequence). After reordering, only a single value at the specific position is of interest. In bit-wise elimination, values that are certain not to be in this position are removed from the sequence.

Values are compared bit-wise starting from the highest order bits. Lets assume that the r-1 highest order bits have already been analysed by this method. Then, all the values that remain for consideration must have the high order r-1 bits equal to each other (and to the result under evaluation). Values that had these bits different were eliminated leaving only n' values in the sequence. The position also has to be adjusted from initial k to k' after eliminating values that were greater. The rth bit of result is determined as $P_{n'}^{k'}(r)$ by ordering the corresponding bits of the reduced sequence and considering k'-position. All the values that differ on the rth bit from $P_{n'}^{k'}(r)$ are eliminated and n' is modified accordingly. If the eliminated values are greater than the quantile bit, then k' is also modified.

The algorithm ends when there is only one value left (or all the values left are equal to each other).

The approach, with changing k and n is not well suited for circuit implementations. Instead of eliminating the values, it is more convenient to modify them in a way that guarantees not to change the result[6,7,8]. If one knows that a value is larger than the k-quantile, all its lower bits are set to 1. If one knows that it is smaller, the lower bits are set to 0. Thus, single bit voting may still be used and the values of k and n are fixed. This is the method used for the presented

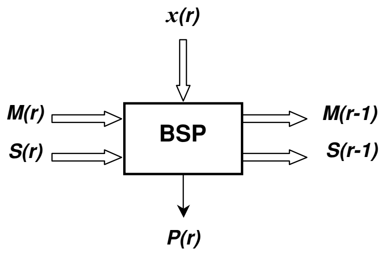


Figure 21-1. Bit-slice processor.

FPGA implementations. It can be formally described by the following iterative equations, where iterations start with the highest order bits ($r = m - 1$) and end with the lowest ($r = 0$):

$$\begin{aligned}
&M_i(r-1) = M_i(r) \vee \left(P_n^k(r) \oplus x_i(r)\right),\\
&S_i(r-1) = (M_i(r) \wedge S_i(r)) \vee (!M_i(r) \wedge x_i(r)),\\
&M_i(n) = S_i(n) = 0, i = 0..n-1,\\
&P_n^k(r) = \sum_{i=0..n-1} \{(!M_i(r) \wedge x_i(r)) \vee (M_i(r) \wedge S_i(r))\} > k
\end{aligned} \tag{1}$$

where
$x_i(r)$ is the r-th bit of i-th pixel in the filtering window,
$P_n^k(r)$ is the r-th bit of the value at k-th position (k-quantile),
$M_i(r)$ and $S_i(r)$ are the modifying functions.

Using the presented Eqs. (1) a bit-slice processor may be implemented (Fig. 21-1). This is a combinatorial circuit that processes the single bits of the input pixels to produce a single bit of the result. The most important part of this bit-slice is the thresholding function corresponding to the last of Eqs. (1).

3. PIPELINED FILTER IMPLEMENTATIONS

The simplest hardware implementation of the filter can be obtained by using m bit-slice processors with connected modifying function inputs and outputs. This would be a fully combinatorial implementation with very long delays, as the modifying functions have to propagate from the highest to the lowest order bits.

Inserting pipelining registers between the bit-slice processors shortens the propagation paths[11]. The registers may be inserted either between all the processors, as shown in Fig. 21-2, or only between some of them. Since this introduces latency between the bit evaluations, additional shift registers are needed on the inputs and outputs to ensure in-phase results.

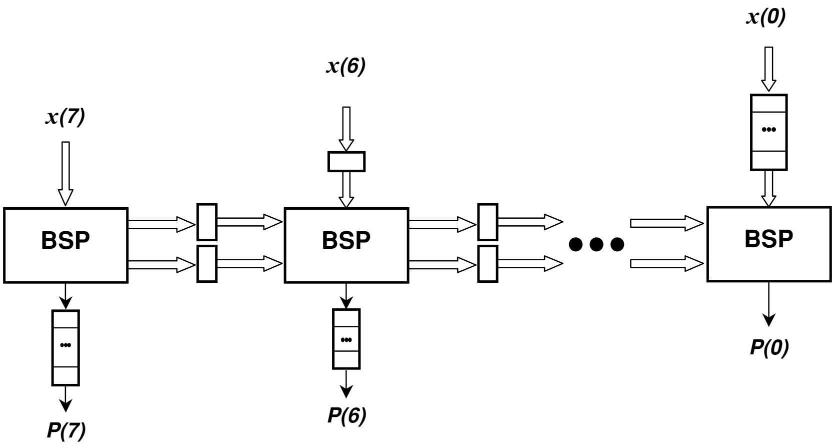


Figure 21-2. Pipeline filter architecture.

This architecture has very short propagation paths between registers and hence ensures highest pixel processing rates. There is latency between the input signals and the output equal to the number of bits in the pixel representations. Normally, in image processing applications this is not a problem. Just the image synchronisation signals need to be shifted correspondingly. It may be unacceptable, though, if image filtering is just a stage in a real-time control application.

4. FPGA IMPLEMENTATION RESULTS

The pipelined filter architecture was implemented for a filtering window of 3×3 pixels. The inputs of the filter were 3 pixel streams: one obtained by scanning the image and two delayed (by one and two horizontal scan periods). The consecutive horizontal window values were obtained by registering the input streams within the filter (to reduce the demand on input/output pads).

All the presented results were obtained by implementing the filter that computed the median. This has no significant effect on the device performance or complexity, except that the min and max filters have much simpler thresholding functions.

The filters were implemented using different size of pixel value representations (binary values of 4, 8, 12, and 16 bits). In each case the smallest and fastest device that could contain the circuit was chosen for implementation. Table 21-1 shows the results of filter implementations using XC4000E family of devices, whereas Table 21-2 presents those for the Virtex-2, and Table 21-3 those for the Spartan-3 packages.

Table 21-1. Filter implementations using XC4000E devices

Pixel representation	Device	Used CLBs	Pixel rate
4 bits	4003EPC84-1	94	55.9 MHz
8 bits	4008EPC84-1	288	50.1 MHz
12 bits	4020EHQ208-1	645	50.6 MHz
16 bits	4025EPG223-2	993	37.5 MHz

Table 21-2. Filter implementations using Virtex-2 devices

Pixel representation	Device	Used slices	Pixel rate
4 bits	2V40FG256-4	99	88.8 MHz
8 bits	2V40FG256-4	209	91.9 MHz
12 bits	2V80FG256-4	345	88.7 MHz
16 bits	2V80FG256-4	453	83.7 MHz

Table 21-3. Filter implementations using Spartan-3 devices

Pixel representation	Device	Used slices	Pixel rate
4 bits	3S50TQ144-5	93	120.4 MHz
8 bits	3S50TQ144-5	217	118.3 MHz
12 bits	3S50TQ144-5	341	119.6 MHz
16 bits	3S50TQ144-5	465	114.1 MHz

The circuit complexity, expressed in terms of the number of cells used (CLBs or Virtex slices), results from the number and complexity of bit-slice processors (complexity of the combinatorial logic) and from the number of registers used in pipelining. The first increases linearly with the size of pixel representation. On the other hand the number of registers used in pipelining increases with the square of this representation. In case of the XC4000 architecture, the pixel representation of 8 bits is the limit, above which the complexity of circuit is determined solely by the pipelining registers (all the combinatorial logic fits in the lookup tables of cells used for pipelining).

The synthesis tools had problems in attaining optimal solutions for the synthesis of thresholding functions in the case of the cells implemented in XC4000 devices (this was not an issue in case of min and max positional filters). Most noticeably, the design obtained when the threshold function was described as a set of minterms required 314 CLBs in case of 8-bit pixel representation. By using a VHDL description that defined the function as a network of interconnected 4-input blocks, the circuit complexity was reduced to the reported 288 cells. The reengineered threshold function had a slight effect on the complexity of the 12-bit filter and none on the 16-bit one.

The most noticeable improvement in using the Virtex-2 devices for positional filter implementations was in the operation speed: approximately 50 MHz in case of the XC4000E devices and 80–90 MHz in case of Virtex-2. Some other architectural improvements are also apparent. The increased functionality of Virtex slices led to much more effective implementations of pipelining registers: the FPGA Express synthesizer implemented them as shift registers instead of unbundled flip-flops, significantly reducing the slice usage. Improved lookup table functionality eliminated the problem of efficient decomposition of threshold function, as well (at least in the case of the 3×3 filtering window).

The results obtained for the Spartan-3 family were similar to the corresponding Virtex-2 ones. The improved performance resulted from higher speed grades of the devices in use. It should be noted that these results were obtained using the synthesis tools integrated within Xilinx ISE 6.2i package, since the available version of FPGA Express could not handle the devices. The synthesis results do not vary significantly, being in some cases better and in others worse in the resultant slice count.

5. CONCLUSIONS

The presented implementation results show that FPGA devices have attained the speed grades that are more than adequate for implementing positional image filters of very high resolution. Furthermore, it is no longer necessary to interconnect multiple FPGA devices or limit the circuit complexity by reducing the pixel representations. In fact, the capabilities of Virtex-2 and Spartan-3 devices exceed these requirements both in terms of performance and cell count.

The proposed bit-wise elimination algorithm with pipelining is appropriate for the cell architecture of FPGA devices. The only problem is the latency, which may be too high in case of long pixel representations. By limiting the pipelining to groups of 2, 3, or more bit-slice processors it is possible to trade off latency against performance.

Positional filtering is just a stage in complex image processing. The analysed filter implementations leave a lot of device resources unused. This is so, even in the case of XC4000E packages, where the cell utilization for representations of 8 bits or more is between 60 and 97%. The cells are mostly used for registering, and the lookup tables are free. These may well be used to implement further stages of image processing.

It is very important that the considered implementations were directly obtained by synthesis from functional descriptions, expressed in VHDL language. This makes feasible the concept of reconfigurable filters, where the user describes the required filtering algorithms in a high-level language, and these are programmed into the filter. However, the design tools have not yet reached the desirable degree of sophistication and reliability. This is especially true of the obscure template matching rules, peculiar to specific synthesis tools. Also, the correctness by design paradigm is not always met – some errors of improperly matched templates were detected only by testing the synthesized device.

REFERENCES

1. M. Juhola, J. Katajainen, T. Raita, Comparison of algorithms for standard median filtering. *IEEE Transactions on Signal Processing*, **39** (1), 204–208 (1991).
2. D.S. Richards, VLSI Median filters, *IEEE Transactions on Acoustics, Speech and Signal Processing*, **38** (1), 145–153 (1990).
3. S. Ranka, S. Sahni, Efficient serial and parallel algorithms for median filtering. *IEEE Transactions on Signal Processing*, **39** (6), 1462–1466 (1991).
4. J.P. Fitch, E.J. Coyle, N.C. Gallagher, Median filtering by threshold decomposition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **32** (6), 553–559 (1984).
5. M.O. Ahmad, D. Sundararajan, A fast algorithm for two-dimensional median filtering. *IEEE Transactions on Circuits and Systems*, **34** (11), 1364–1374 (1987).
6. C.L. Lee, C.W. Jen, Binary partition algorithms and VLSI architecture for median and rank order filtering. *IEEE Transactions on Signal Processing*, **41** (9), 2937–2942 (1993).
7. C.-W. Wu, Bit-level pipelined 2-D digital filters for real-time image processing. *IEEE Transactions on Circuits and Systems for Video Technology*, **1** (1), 22–34 (1991).
8. D. Caban, J. Jarnicki, A reconfigurable filter for digital images processing (in Polish). *Informatyka*, **6**, 15–19 (1992).
9. D. Caban, Hardware implementations of a real time positional filter. In: *Proceedings of 5th Microcomputer School Computer Vision and Graphics*, Zakopane, pp. 195–200 (1994).
10. S.C. Chan, H.O. Ngai, K.L.Ho, A programmable image processing system using FPGAs. *International Journal Electronics*, **75** (4), 725–730 (1993).
11. D. Caban, W. Zamojski, Median filter implementations. *Machine Graphics & Vision*, **9** (3), 719–728 (2000).

Chapter 22

A METHODOLOGY FOR DEVELOPING IP CORES THAT REPLACE OBSOLETE ICS

An industrial experience

Wojciech Sakowski[1], Mirosław Bandzerewicz[2], Maciej Pyka[2], and Włodzimierz Wrona[3]

[1]Institute of Electronics, Silesian University of Technology, ul. Akademicka 16, 44-100 Gliwice, Poland; e-mail: sak@boss.iele.polsl.gliwice.pl

[2]Evatronix S.A., ul. Dubois 16, 44-100 Gliwice, Poland; e-mail: mirek@gliwice.evatronix.com.pl

[3]Technical University of Bielsko-Biała, Department of Electrical Engineering, ul. Willowa 2, 43-308 Bielsko-Biała, Poland

Abstract: This paper presents a proven methodology of the development and productization of virtual electronic components. The methodology consists of rigorous approach to the development of component specification, reverse engineering of behavior of reference circuits by means of hardware simulator, application of industrystandard rules to coding of RTL model in a hardware description language, and extensive testing and verification activities leading to high quality synthesizable code and to working FPGA prototype. In the final stage called productization a series of deliverables are produced to ensure effective reuse of the component in different (both FPGA and ASIC) target technologies.

Key words: Virtual Components; IP Cores; Hardware Description Languages; high level design; quality assurance.

1. OBJECTIVE AND MOTIVATION

The objective of the effort described in this paper was to define a quality assurance policy for the development of virtual components based on existing integrated circuits. At the time this policy was developed, our company specialized in the development of IP cores compatible to 8-bit and 16-bit microcontrollers and microprocessors. Some of these cores ensure cycle level compatibility needed for direct obsolete chip replacement. Other cores are merely instruction-set compatible and offer architectural improvements over original chips[1]. They are aimed at providing functionality of original parts in systems-on-chip, where original pin timing behavior is not required.

Our approach is based on the methodology recommended in Ref. 2, but it reflects to some extent peculiarities of our profile as well as the fact that we have no access to certain EDA tools recommended in Ref. 2. We found inspiration in the paper presented by SICAN company (now SCI-WORX) at the FDL'99 in Lyon[3].

The main motivation for the definition of a formalized methodology was to assure a high quality of the cores that we develop. Our first experiences in the development of a microcontroller core compatible to Intel 8051 chip[1] showed that lack of consistent and rigorous methodology results in a buggy core. Moreover, lack of a clear and complete specification turns the debugging of our first core into a nightmare.

2. OVERVIEW OF THE METHODOLOGY

2.1 Design flow

Basic development steps in the creation of a virtual component include:

- Development of the macro specification,
- Partitioning the macro into subblocks,
- Development of a testing environment and a test suite,
- Design and verification of subblocks,
- Macro integration and final verification,
- Prototyping the macro in FPGA,
- Productization.

We will discuss these stages one by one in this chapter, focusing on details related to our experiences. In addition we will present in more detail the use of hardware modeling in the specification and verification process of virtual components.

2.2 Project management issues

At the beginning of a new project all the steps enumerated earlier are refined into subtasks and scheduled. Human and material resources are allocated to the project. Usually, several projects are being realized parallelly. Therefore people, equipment, and software, have to be shared among these projects. We use MS Project software to manage scheduling of tasks and allocation of resources.

3. DEVELOPMENT OF THE MACRO SPECIFICATION

We use the documentation of an original device as a basis for the specification of the core modeled after it. However, the documentation provided by the chip manufacturer is oriented toward chip users and does not usually contain all details of chip behavior that are necessary to recreate its full functionality. Therefore analysis of the original documentation results in a list of ambiguities, which have to be resolved by testing the original chip. The overall testing program is usually very complex, but the first tests to be written and run on a hardware modeler (see point 5) are those that resolve ambiguities in the documentation.

At a later stage of specification we use an Excel spreadsheet to document all operations and data transfers that take place inside the chip. Spreadsheet columns represent time slots and rows represent communication channels. Such an approach enables gradual refinement of scheduling of data transfers and operations up to the moment when clock-cycle-accuracy is reached. It reveals potential bottlenecks of the circuit architecture and makes it easy to remove them at an early design stage.

4. PARTITIONING INTO SUBBLOCKS

The dataflow spreadsheet makes it easier to define proper partitioning of the macro into subblocks. This first level of design hierarchy is needed to handle the complexity and for easier distribution of design tasks between several designers. The crucial issue in this process is distribution of functions between the subblocks, definition of the structural interfaces, and specification of timing dependencies between them.

5. ROLE OF THE HARDWARE MODELING IN THE VIRTUAL COMPONENT DEVELOPMENT PROCESS

5.1 Introduction to hardware modeling

By hardware modeling we understand the use of real chips as reference models inside a simulated system (which contains them). At the turn of the 1980s and 1990s, hardware modeling was used in a board-level system simulations due to the lack of behavioral models of LSI/VLSI devices used in those days for package construction. Racal-Redac's CATS modeler dates back to those days. Together with a CADAT simulator running on a SUN workstation they form an environment that allows the user to simulate systems that consist of integrated circuits for which no behavioral models are available. These circuits are modeled by real chips and may interact within the mentioned environment with software-based testbench and other parts for which simulation models exist.

The hardware modeler proved to be useful in our company during specification and verification stages. The greatest role of hardware models is related to reverse engineering, when the goal is documenting functionality of existing catalogue parts (usually obsolete). Reverse engineering is essential during development of the specification of IP cores meant to be functionally equivalent to those parts. Hardware modeling resolves many ambiguities, which are present in the referenced chip documentation.

The hardware modeler is also useful for testing the FPGA prototypes of virtual components, independent of whether it was used during the specification stage.

5.2 Testing the reference chip

As a reference for our virtual components we use hardware models that run on a (second hand) CATS hardware modeler (Fig 1). The hardware modeler is connected via network to the CADAT simulator. The environment of the chip is modeled in C. Test vectors supplied from a file may be used for providing stimuli necessary to model interaction of the modeled chip with external circuits (e.g., interrupt signals).

An equivalent testing environment is developed in parallel as a VHDL testbench to be run on a VHDL simulator. We use Aldec's Active-HDL simulator, which proved to be very effective in model development and debugging phase.

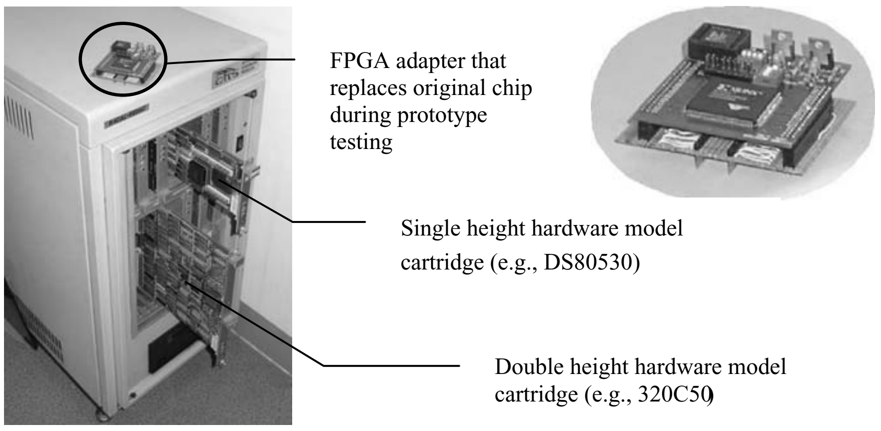

FPGA adapter that replaces original chip during prototype testing
Single height hardware model cartridge (e.g., DS80530)
Double height hardware model cartridge (e.g., 320C50)

Figure 22-1. CATS hardware modeler.

It enables the import of the testing results obtained with a hardware model into its waveform viewer in order to compare them with simulated behavior of the core under development.

5.3 Improvements of the hardware modeling technology

The growth of requirements for hardware models comes as a consequence of developing of more and more complex virtual components. This situation brings problems such as too many pins of the reference chip, system overload due to many simultaneous simulations, long-lasting preparation of a new model, and lack of ability to model some physical features (e.g., bi-directional asynchronous pins). The fact that definition of simulation environment demands skill in using an exotic BMD language and unusual simulation environment is also a severe restriction.

The solution to these problems is the new system developed at Evatronix in 2003–2004 under the name of *Personal Hardware Modeler (PHM)*[4]. This name reflects the basic feature which is the transformation of a huge centralized workstation into a light desktop device, which may be connected to any PC machine. From the user's point of view the new system is easy enough to allow the preparation of a new hardware model by any engineer without having any specialized knowledge of hardware modeling. The biggest effort is now reduced to the design and manufacturing of an adapter-board connecting the reference chip to the modeler.

The rule of operation of the modeler is similar to the CATS system mentioned before. It is based on periodic stimulation of the reference chip followed by its response detection (called dynamic modeling). The PHM device is built upon an FPGA circuit containing serial communication interface to the PC and other logic that performs stimulation and response detection on a reference IC. PHM stores stimulation vectors in local memory, applies a sequence of them to all input pins of the device under test in real-time, and sends detected responses back to the PC, where they are processed by VHDL simulator.

The goal of the whole system is to substantially improve the reference circuit examination process, test suite development, and also VC prototype verification.

6. VERIFICATION PROCESS

6.1 Test suite development

Test suite development is based on specification. Specification is analyzed and all the functional features of the core that should be tested for the original device are enumerated. The test development team starts with development of tests that are needed to resolve ambiguities in the available documentation of the chip to which a core has to be compliant.

Most of the functional tests are actually the short programs written in the assembly language of the processor that is modeled. Each test exercises one or several instructions of the processor. For instructions supporting several addressing modes, tests are developed to check all of them. After compiling a test routine the resulting object code is translated to formats that may be used to initialize models of program memory in the testbenches (both in CADAT and VHDL environments). We have developed a set of utility procedures that automate this process.

In order to test processor interaction with its environment (i.e., I/O operations, handling of interrupts, counting of external events, response to reset signal) a testbench is equipped with a stimuli generator.

6.2 Code coverage analysis

The completeness of the test suite is checked with code coverage tool (VN-Cover from TransEDA). The tool introduces monitors into the simulation environment and gathers data during a simulation run. Then the user can check how well the RTL code has been exercised during simulation. There are a number of code coverage metrics that enable analysis of how good the test suite is, what parts of the code are not properly tested, and why. At Evatronix we make use of the following metrics: statement coverage, branch coverage, condition/expression coverage, and path and toggle coverage.

The *statement coverage* shows whether each of the executable code statements was actually executed and how many times. It seems obvious that statement coverage below 100% indicates that either some functionality was not covered by the test suite, or untested code is unnecessary and should be removed. *Branch coverage* may reveal why a given part of the code is untested. It checks whether each branch in *case* and *if-then-else* statements is executed during simulation. As some branches may contain no executable statements (as e.g., *if-then* statement with no *else* clause), it may happen that branch coverage is below 100% even if statement coverage reaches this level. Analysis why the given branch is not taken is simplified with availability of *condition coverage* metrics. With this metric one may analyze whether all combinations of subexpressions that form branch conditions are exercised. *Path coverage* shows whether all possible execution paths formed by two subsequent branch constructs are taken. *Toggle coverage* shows whether all signals toggled from 0 to 1 and from 1 to 0. We target 100% coverage for all these metrics. In addition we also use *FSM coverage* (*state, arc*, and *path*) metrics to ensure that control parts of the circuit are tested exhaustively.

Incompleteness of the test suite may result in leaving bugs in untested parts of the code. On the other hand code coverage analysis also helps to reveal (and remove) redundancy of the test suite.

6.3 Automated testbench

Our cores are functionally equivalent to the processors they are compliant to, but they are not always cycle accurate. Therefore a strategy for automated comparison of results obtained with hardware modeler to those obtained by simulating RTL model was developed.

Scripts that control simulators may load the program memory with subsequent tests and save the simulation data into files. These files may serve as reference for postsynthesis and postlayout simulation. The testbench that is used for these simulation runs contains a comparator that automatically compares simulator outputs to the reference values.

7. SUBBLOCK DEVELOPMENT

The main part of the macro development effort is the actual design of subblocks defined during specification phase. At the moment we have no access to tools that check the compliance of the code to a given set of rules and guidelines. We follow the design and coding rules defined in Ref. 1. We check the code with VN-Check tool from TransEDA to ensure that the rules are followed. Violations are documented.

For certain subblocks we develop separate testbenches and tests. However, the degree to which the module is tested separately depends on its interaction with surrounding subblocks. As we specialize in microprocessor core development it is generally easier to interpret the results of simulation of the complete core than to interpret the behavior of its control unit separated from other parts of the chip. The important aspect here is that we have access to the results of the test run on the hardware model that serves as a reference.

On the other hand certain subblocks like arithmetic-logic unit or peripherals (i.e., Universal Asynchronous Receiver/Transmitters (UARTs) and timers) are easy to test separately and are tested exhaustively before integration of the macro starts.

Synthesis is realized with tools for FPGA design. We use Synplify, FPGA Express, and Leonardo. We realize synthesis with each tool looking for the best possible results in area-oriented and performance-oriented optimizations.

8. MACRO INTEGRATION

Once the subblocks are tested and synthesized they may be integrated. Then all the tests are run on the RTL model and the results are compared with the hardware model. As soon as the compliance is confirmed (which may require a few iterations back to subblock coding and running tests on integrated macro again) a macro is synthesized towards Xilinx and Altera chips and the tests are run again on the structural model.

9. PROTOTYPING

The next step in the core development process is building of a real prototype that could be used for testing and evaluation of the core.

At present we target two technologies: Altera and Xilinx. Our cores are available to users of Altera and Xilinx FPGAs through AMPP and AllianceCORE programs. Shortly we will implement our cores in Actel technologies, as well. Placing and routing of a core in a given FPGA technology is realized with vendor-specific software. The tests are run again on the SDF-annotated structural model. We developed a series of adapter boards that interface FPGA prototype to a system in which a core may be tested or evaluated.

The simplest way to test the FPGA prototype is to replace an original reference chip used in the hardware modeler with it. This makes it possible to compare the behavior of the prototype with the behavior of the original chip. However, for some types of tests even a hardware modeler does not provide the necessary speed. These tests can only be executed in a prototype hardware system at full speed. Such an approach is a must when one needs to test a serial link with a vast amount of data transfers, or to perform floating point computations for thousands of arguments. Our experience shows that even after an exhaustive testing program, some minor problems with the core remains undetected until it runs a real-life application software.

For this reason we have developed a universal development board (Fig. 22-2). It can be adapted to different processor cores by replacement of onboard programmable devices and EPROMs. An FPGA adapter board (see Fig. 22-1) containing the core plugs into this evaluation board. An application program may be uploaded to the on-board RAM memory over a serial link from PC. Development of this application program is done by a separate design team. This team actually plays a role of an internal beta site, which reveals problems in using the core before it is released to the first customer.

The FPGA adapter board can also be used to test the core in the application environment of a prototype system. Such system should contain a microcontroller or microprocessor that is to be replaced with our core in the

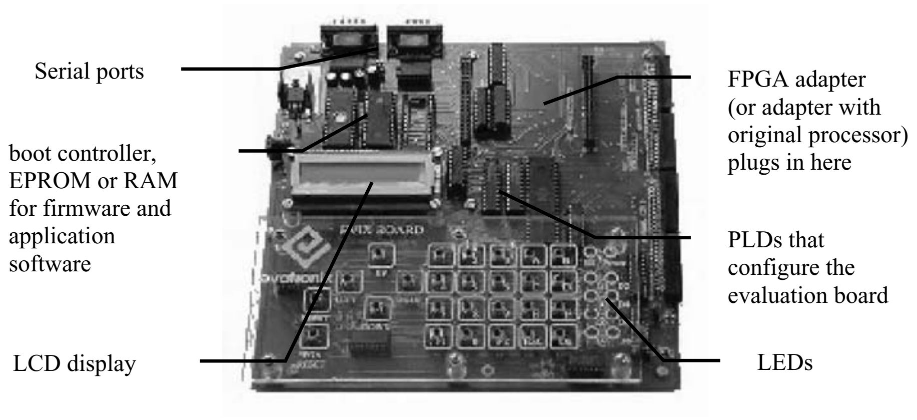

Serial ports
boot controller, EPROM or RAM for firmware and application software
LCD display
FPGA adapter (or adapter with original processor) plugs in here
PLDs that configure the evaluation board
LEDs

Figure 22-2. Development boards for testing processor core.

integrated version of the system. The adapter board is designed in such a way that it may be plugged into the microprocessor socket of the target system. Using this technique we made prototypes of our cores run into ZX Spectrum microcomputer (CZ80cpu core) and SEGA Video Game (C68000 core), in which they replaced original Zilog® and Motorola® processors.

10. PRODUCTIZATION

The main goal of the productization phase is to define all deliverables that are necessary to make the use of the virtual component in the larger design easy. We develop and run simulation scripts with Modelsim and NC Sim simulators to make sure that the RTL model simulates correctly with them.

While we develop cores in VHDL we translate them into Verilog, to make them available to customers who only work with Verilog HDL. The RTL model is translated automatically while the testbench manually. The equivalence of Verilog and VHDL versions is exhaustively tested.

Synopsys Design Compiler scripts are generated with the help of the FPGA Compiler II. Synthesis scenarios for high performance and for minimal cost are developed.

For FPGA market an important issue is developing all the deliverables required by Altera and Xilinx from their partners participating in *AMPP* and *AllianceCore* third party IP programs.

User documentation is also completed at productization stage (an exhaustive, complete, and updated specification is very helpful when integrating the core into a larger design).

11. EXPERIENCES

The methodology described in this paper was originally developed in the years 1999 and 2000 during the design of a few versions of 8051-compatible microcontroller core[1]. It was then successfully applied to the development of IP cores compatible to such popular chips as Microchip PIC® 1657 microcontroller, Motorola 68000 16-bit microprocessor and 56002 digital signal processor, Zilog® Z80 8-bit microprocessor and its peripherals, TI® 32C025 dsp and Intel® 80186 16-bit microcontroller.

After accommodating certain improvements of this methodology, we documented it in our quality management system which passed the ISO 9001 compliance audit in 2003. Presently we are looking at complementing it with functional coverage and constrained random verification techniques.

REFERENCES

1. M. Bandzerewicz, W. Sakowski, Development of the configurable microcontroller core. In: *Proceedings of the FDL'99 Conference*, Lyon (1999).
2. M. Keating, P. Bricaud, *Reuse Methodology Manual* (2nd ed.). Kluwer Academic Publishers (1999).
3. J. Haase, Virtual components – From research to business. In: *Proceedings of the FDL'99 Conference*, Lyon (1999).
4. Maciej Pyka, Wojciech Sakowski, Włodzimierz Wrona, Developing the concept of hardware modeling to enhance verification process in virtual component design. In: *Proceedings of IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, Poznan (2003).

Index