Studies in Systems, Decision and Control 45

Andrei Karatkevich
Arkadiusz Bukowiec
Michał Doligalski
Jacek Tkacz *Editors*

Design of Reconfigurable Logic Controllers

Studies in Systems, Decision and Control

Volume 45

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

About this Series

The series "Studies in Systems, Decision and Control" (SSDC) covers both new developments and advances, as well as the state of the art, in the various areas of broadly perceived systems, decision making and control- quickly, up to date and with a high quality. The intent is to cover the theory, applications, and perspectives on the state of the art and future developments relevant to systems, decision making, control, complex processes and related areas, as embedded in the fields of engineering, computer science, physics, economics, social and life sciences, as well as the paradigms and methodologies behind them. The series contains monographs, textbooks, lecture notes and edited volumes in systems, decision making and control spanning the areas of Cyber-Physical Systems, Autonomous Systems, Sensor Networks, Control Systems, Energy Systems, Automotive Systems, Biological Systems, Vehicular Networking and Connected Vehicles, Aerospace Systems, Automation, Manufacturing, Smart Grids, Nonlinear Systems, Power Systems, Robotics, Social Systems, Economic Systems and other. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution and exposure which enable both a wide and rapid dissemination of research output.

More information about this series at http://www.springer.com/series/13304

Andrei Karatkevich · Arkadiusz Bukowiec
Michał Doligalski · Jacek Tkacz
Editors

Design of Reconfigurable Logic Controllers

Editors
Andrei Karatkevich
Faculty of Computer Science, Electrical Engineering and Automatics
University of Zielona Góra
Zielona Góra
Poland

Michał Doligalski
Faculty of Computer Science, Electrical Engineering and Automatics
University of Zielona Góra
Zielona Góra
Poland

Arkadiusz Bukowiec
Faculty of Computer Science, Electrical Engineering and Automatics
University of Zielona Góra
Zielona Góra
Poland

Jacek Tkacz
Faculty of Computer Science, Electrical Engineering and Automatics
University of Zielona Góra
Zielona Góra
Poland

ISSN 2198-4182 ISSN 2198-4190 (electronic)
Studies in Systems, Decision and Control
ISBN 978-3-319-26723-4 ISBN 978-3-319-26725-8 (eBook)
DOI 10.1007/978-3-319-26725-8

Library of Congress Control Number: 2015956377

© Springer International Publishing Switzerland 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by SpringerNature
The registered company is Springer International Publishing AG Switzerland

Preface

Digital design of control circuits is a very important part of computer science and electronics, and its importance has increased in recent years. Nowadays, digital systems are widely present in everyone's life and they are a part of our existence. The rapid growth of silicon technology is being observed, and it causes the augmentation of the controlled systems such as data paths in digital devices, peripheral devices of the computers or the industrial electromechanical processes for which the programmable controllers are used. It causes that need for more complex and faster control units noticeable, and new design methodologies of such complex systems are required.

In this book, we present the research activities and achievements in the area of design of reconfigurable control circuits of several research teams from different countries (Poland, Belarus and Portugal) as well as the historical perspective of development of some aspects of logical control technology. The chapters of the book cover different fields of the topic, from control system specification and design to synthesis and verification. The important question of cooperation between control unit and data path is also discussed. The book focuses first of all on the parallelism in logical control, taking into account complexity of the systems under control, an unavoidable element of modern logical control algorithms. Reconfigurability is another important aspect of the approaches presented in the book; nowadays the control systems often have to be flexible, hence possibility of their partial reconfiguration during runtime is very essential. As the models of parallel control algorithms, the interpreted Petri nets and concurrent generalizations of finite-state machines are used. Various kinds of UML diagrams are used at different steps of design processes for specification and modelling. The described methodologies mostly suppose the FPGA realization of the reconfigurable control devices.

The editors of this book hope that it will be a valuable reading for both researches and students of computer science and electronics, and engineers working in the area of design of digital control and embedded systems. The reader is presumed to have a basic knowledge of digital design, automata theory and Petri nets.

Zielona Góra, Poland
August 2015

Andrei Karatkevich
Arkadiusz Bukowiec

Contents

Petri Nets in Design of Control Algorithms 1
Andrei Karatkevich

Synthesis and Implementation of Parallel Logic Controllers in All Programmable Systems-on-Chip 15
Valery Sklyarov, Iouliia Skliarova and João Silva

Circuit Implementation of Parallel Logical Control Algorithms Represented in PRALU Description 31
P.N. Bibilo, Yu.V. Pottosin, V.I. Romanov and A.D. Zakrevskij

Effective Partial Reconfiguration of Logic Controllers Implemented in FPGA Devices 45
Remigiusz Wiśniewski, Monika Wiśniewska and Marian Adamski

An Application of Logic Controller for the Aerosol Temperature Stabilization 57
Michał Doligalski, Marek Ochowiak and Anna Gościniak

Symbolic Coloring of Petri Nets 67
Jacek Tkacz

Modular Synthesis of Petri Nets 77
Jacek Tkacz and Marian Adamski

Architectural Synthesis of Petri Nets 93
Arkadiusz Bukowiec

Decomposition-Based Methods for FSM Implementation 103
Mariusz Rawski, Piotr Szotkowski and Paweł Tomaszewicz

Using UML Behavior Diagrams for Graphical Specification of Programs for Logic Controllers 131
Grzegorz Bazydło and Marian Adamski

Various Interpretations of Actions of UML Activity Diagrams in Logic Controller Design 143
Michał Grobelny, Iwona Grobelna and Marian Adamski

Model Checking of UML Activity Diagrams Using a Rule-Based Logical Model 153
Iwona Grobelna, Michał Grobelny and Marian Adamski

UML Support for Statecharts-Based Digital Logic Controller Design in FPGA Technology 165
Grzegorz Łabiak

Index 181

Petri Nets in Design of Control Algorithms

Andrei Karatkevich

Abstract The chapter presents an overview of applying the Petri nets as a model and a way of specification of the parallel logical control algorithms. The history of using the Petri nets for representing the structures of the parallel control algorithms is presented. The extensions of the Petri net model applied in the area of logical control are discussed. The Petri net-based programming languages used for programmable logic controllers, such as SFC, GRAFCET or PRALU, are considered.

Keywords Petri nets · Logic controllers · Specification · FPGA · Parallel control algorithms

1 Introduction

There are known various ways of specification of logical control algorithms in general and programs for logic controllers in particular. The algorithms of continuous control can be represented in form of the differential equations [1]; if we limit ourselves to the logical control algorithms, we can find out that such algorithms can be described by means of the textual programming languages (usually the dedicated ones, like Structured Text [2] or SystemC [3]) or graphical languages and diagrams (Harel statecharts [4], LD, SFC, FBD [2], CFC [5], ASM [6], flowcharts [7], state diagrams [8] and so on). The mathematical models belhind those languages are first of all Boolean functions, Finite State Machines (with their generalizations, enhanced with, among others, concurrency and hierarchy [9–11]) and also the Petri nets. The most known (but not the only one) Petri net-based language for specification of logical control algorithms is Sequential Function Chart (SFC) [2, 12].

The Petri nets and Petri net-based languages are not very popular among the engineers. However, they have some unique advantages. The Petri nets allow to

A. Karatkevich (✉)
Institute of Electrical Engineering,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: a.karatkevich@iee.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_1

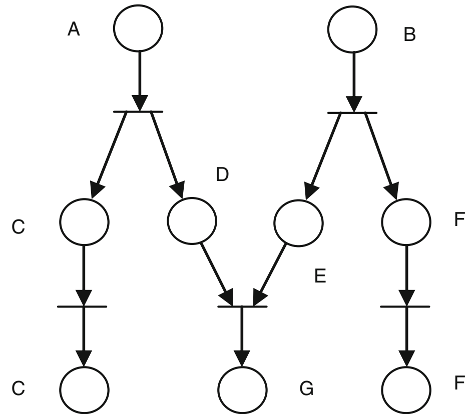


Fig. 1 An example of parallel processes described by a Petri net

describe directly the parallel threads of an algorithm, and structure of the parallelism can be more flexible than most of programming languages (especially the languages for programmable logic controllers) and the FSM-based models (such as HCFSM or UML state machines) allow. It means that a Petri net allows to specify a parallel algorithm beyond the traditional fork-join model, in which for each "fork" there should be just one "join". Instead, using a Petri net-based language one can describe, for example, a situation in which, having two parallel threads A and B, thread A divides into parallel threads C and D, B divides into E and F, and then threads D and E merge into thread G, which is executed further in parallel with threads C and F (Fig. 1). On the other hand, there is a wide range of the methods of formal analysis of Petri nets [13, 14], which can be used for verification of the Petri net based algorithms.

The chapter is organized as follows. Section 2 presents a brief review of the history of applying Petri nets to specification of logical control algorithms. Section 3 describes the main mathematical models of the extended Petri nets used in the discussed area. Section 4 presents the most popular Petri net-based language—Sequential Function Chart. Section 5 describes another language for description of parallel algorithms of logical control, PRALU [15]. In Sect. 6 some other approaches to Petri net based specification of control algorithms are discussed. The final section contains a conclusion.

2 Historical Review

Petri nets in their classical form are autonomous, i.e. they do not communicate with the external world. However, the C. A. Petri's dissertation [16], which introduced the notion, has the title "Communication with Automata", and according to the vision presented in it a Petri net is an abstract model of an automata network, which means

that the Petri nets from the very beginning were considered as a model of a system which does communicate with its environment. The Petri nets are understood as "a general purpose mathematical model for describing relations existing between conditions and events" [17].

Nevertheless, the Petri net-based models extended in such a way that they could communicate by means of the binary signals and represent the logical control algorithms, were not described (to the best of our knowledge) before the second half of 1970s. At that time appeared the concept of the Petri nets extended in such a way that the conditions, depending on the input signals, are associated with the transitions, and the actions, setting the values of the output signals, are associated with the places or, in some cases, with the transitions. Such a model [18, 19] is referred as the *Marking Diagrams* (MDs) [20, 21], *interpreted Petri nets* [12, 22, 23] or *control interpreted Petri nets* [24] (the last two notions are, strictly speaking, wider; numerous Petri net extensions are mentioned as the "interpreted Petri nets" [17], especially a wide range of non-autonomous Petri nets [20]).

Approximately at the same time a language for specification of control algorithms has been developed, which evolved later to an international standard and one of the most popular Petri net-based programming languages for logic controllers (one may say, the *only* popular language of this kind). The language is GRAFCET (an abbreviation of "GRAphe Fonctionnel de Commande Etapes/Transitions" or earlier "GRaphe de l'AFCET") [21, 25]. It was designed in France by the AFCET commission (*Association française pour la cybernétique économique et technique*, later known as *Association des sciences et technologies de l'information*), and the first report officially representing the new language was published in 1977 [26]. In a short time after publishing of the mentioned report, a group in the ADEPA agency (*Agence nationale pour le Développement de la Production Automatisé*) was organized for normalization of the GRAFCET standard to make possible its practical use in programming of the controllers. ADEPA's report [27] was published in 1979 and became a base of most of the standards including GRAFCET. Since that, GRAFCET has been implemented in the engineering practice. It was used and actively supported by the company Telemecanique, at that time one of the world's leading manufacturers of digital controllers (later it was acquired by Schneider Electric).

In 1988 the International Electrotechnical Commission (IEC) has published IEC 848, an international standard defining a GRAFCET-based graphical language. The current (since 1993) international standard for programmable logic controllers, IEC 61131 (the corresponding European norm established by the European Committee for Standardization is EN 61131), defines five programming languages for industrial PLCs. Among them there is the language Sequential Function Chart (SFC), which is directly based on the function charts described in IEC 848 [2, 12]. SFC can be used to structure the internal organization of a program consisting of the sub-programs written in other languages of the standard (usually ST and LD languages) and also can be used in its "pure" form for specifying the logical control algorithms. The latest (third) version of the standard was published in 2013 [28], it also includes SFC.

SFC is supported by numerous programming and modelling platforms for logic controllers [12, 49], such as STEP 7 by Siemens [30] and Control Builder by ABB [31]. It is widely used in SCADA systems.

At the end of 1970s and the beginning of 1980s some other research groups started developing the Petri net-based models and languages for logical control. Such attempts were made in the Soviet Union [32]. A research group led by A. Zakrevskij in the Institute of Engineering Cybernetics of the Belarusian Academy of Science has developed the formal models of parallel control algorithms [33, 34] and the language PRALU ("PRostoy Algoritm Logicheskogo Upravleniya"—"Simple logical control algorithm"). A detailed theory of parallel logical control algorithms arose around that, and the methods of optimized hardware implementation of the PRALU algorithms have been designed [15, 35–39]. Later the converters from PRALU to VHDL ([40], see also the chapter "Circuit implementation of parallel logical control algorithms represented in PRALU description" of this book) and from PRALU to LD [41] were designed.

In the early 1980s research in the similar field started at the Technical University of Zielona Góra, Poland, by the team led by M. Adamski [42–44]. An original model of concurrent state machine was designed. The research concentrated mostly on the methods of hardware implementation of parallel logical control algorithms (PLA, later FPGA implementation) [45, 46]. Three versions of Petri net specification format (PNSF), intended "to describe a Petri net specification of a parallel controller behaviour in textual form", were designed [47, 48]. Further researches in the area of Petri nets in Zielona Góra include also such directions as formal analysis, verification and validation [24, 49–51], modelling of the interpreted Petri nets in the hardware description languages [52–55], some theoretical aspects of Petri nets [56], modelling by Petri nets other models of parallel logic controllers [57] or applying decision diagrams to analysis of Petri nets [58, 59].

The interesting results in applying Petri net models to logical control were obtained at Universidade Nova de Lisboa, Portugal [60–62]. Some of other publications in this area which are worth mentioning are as follows: [63–68]. More references can be found in [13, 21].

3 Interpreted Petri Nets in Logical Control

The basic model of *Petri net* (*PN*) [13] is defined as a tuple $\Sigma = (P, T, F, M_0)$, where P is a finite non-empty set of *places*, T is a finite non-empty set of *transitions*, F is a set of *arcs* such that $F \subseteq (P \times T) \cup (T \times P)$, M_0 is an *initial marking*.

A state of a Petri net, called a *marking*, is defined as a function $M : P \to \mathbb{N}$ (for the safe Petri nets, which are usually used in applications to logical control, $M : P \to \{0, 1\}$). It can be considered as a number of tokens situated in the net places. A place containing a token is called a *marked place*. Sets of input and output places of a transition are defined respectively as follows: $\bullet t = \{p \in P : (p, t) \in F\}$, $t\bullet = \{p \in P : (t, p) \in F\}$. A transition t is *enabled* and can *fire* (be executed), if

$\forall p \in \bullet t : M(p) > 0$. Transition firing removes one token from each input place and adds one token to each output place. A marking can be changed only by a transition firing. It is worth noting that a typical structure of a parallel logical control algorithm is a safe extended free choice net [13] with a single-token initial marking, sometimes referred as an α-net [41, 69].

This model is autonomous and should evidently be enriched by some tools of communication with outer world to make possible specifying the control algorithms by means of it. In the case of logical control, such communication is realized by the logical signals (Boolean variables). The variables can be the input ones (X), meaning values of the signals coming from a controlled system, and the output ones (Y), meaning values of the signals sent by a controller to a controlled system. Sometimes those sets have a non-empty common part $Z = X \cap Y$, then the variables belonging to the set Z are considered to be the internal variables, which can be changed only by the Petri net itself [15].

An essential feature of the basic Petri nets is that an enabled transition *can* fire, but at that level of abstraction it is not defined *when* and *whether* it is going to fire. Of course a model of a logical control algorithm should be deterministic, and conditions of a transition firing should be concretised. In the control interpreted Petri nets, virtually in all of their variants, a condition being a Boolean function (usually an elementary conjunction) of the input variables can be associated with every transition. An enabled transition fires, when the condition is satisfied. If the condition is satisfied at the moment when the transition becomes enabled, it fires immediately. It fires immediately also in the case when no condition is associated with the transition explicitly (then it is supposed that the condition is always satisfied; such transitions are typically used for synchronization of the concurrent processes [64]).

There are two different ways of managing the output variables in the interpreted Petri nets: "Moore type" and "Mealy type" [12]. In the first case a subset of the output variables (maybe empty) is associated with every place of the Petri net. Then a variable obtains value '1', if it is associated with at least one place marked in the current marking, otherwise its value is '0' [64]. In the second case an action being described by a conjunction of the output variables or their negations is associated to every transition [15].

Besides, the interpreted Petri nets can be considered as synchronous [70] or asynchronous [71].

An interpreted Petri net can be understood as a *parallel automaton*—a generalization of a finite state machine, which can be at the same time in one or more *local states* (which correspond to the marked places). A *global state* of such automaton is a set of the local states which are currently active (it corresponds to a marking) [12, 15, 34, 37–39, 41, 43].

Other popular extensions of the Petri net models of the logical control algorithms are the following:

- hierarchy [22, 25, 72–74] (such models include *macroplaces* or rarely *macrotransitions*),

- priorities [25, 76, 77] (a mechanism allowing to decide which of simultaneously enabled conflicting transitions should be executed),
- history attribute [22, 72, 75] (a possibility for a subnet to remember its marking when the corresponding macroplace looses token),
- inhibitor and enabling arcs [12, 22, 78] (an inhibitor arc disables a transition when a place is marked; an enabling arc from a place to a transition should have a token for the transition to be enabled, but does not loose it when the transition is fired),
- reset arcs [17, 38] (a possibility to empty a place or a subset of places when transition fires, independently of their previous state),
- delays [17, 79] (time intervals can be associated with places or transitions).

The very detailed interpreted Petri net models applicable for modelling of logical control algorithms are described in [17, 72, 85].

4 GRAFCET and SFC

Sometimes the terms "GRAFCET" and "Sequential Function Chart" are used as synonyms, in other cases GRAFCET is considered as a basis of SFC [17]. Usually GRAFCET is understood as a technology-independent language for functional specification of sequential control, and it is defined as such by IEC 60848 standard, and SFC, according to its definition by IEC 61131-3 standard, as its implementation and concretization as a programming language for logic controllers.

A program in SFC consists of *steps* and *transitions*. A step can be active or inactive (one or more steps are the *initial* ones and are activated when the program starts). A transition can have one or more input steps and one or more output steps, connected to it by the *directed links*. A condition is associated with every transition. If all input steps of a transition are active, and the transition condition is satisfied, then the transition is fired, which means deactivation of all its input steps and activation of all its output steps. It is easy to see that such model is close to a binary (safe) Petri net. A difference is, that SFC is considered to describe the synchronous systems, hence several transitions can be fired simultaneously, which is not the case for the classical Petri nets.

One or more *actions* can be associated with a step of SFC. An action can be an operation on a logical variable (such as set or reset) or a sub-program specified in SFC (i.e. the SFC programs can be organized hierarchically) or in another language of IEC 61131-3 standard (not all programming environments for PLCs support all of them). An action associated with the step can be executed once or cyclically during the whole time when the step remains active, which depends on the *action qualifier* associated to the action. Other qualifiers allow to set delays of execution of an action, to set the time limits of execution of an action or to deactivate an action activated by another step.

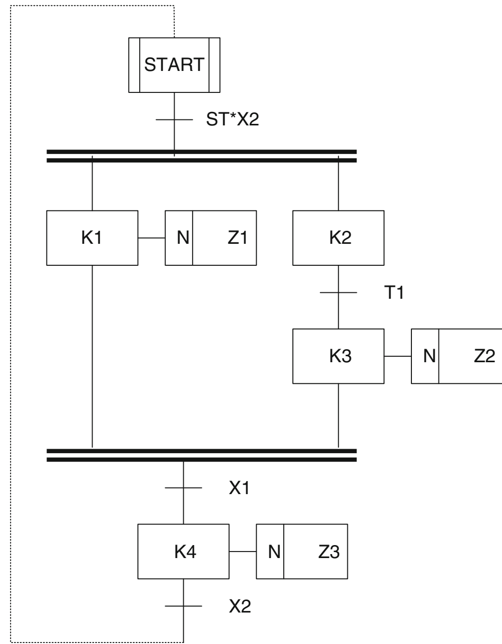


Fig. 2 An example of a control algorithm in SFC (taken from [12])

A simple example of SFC is shown in Fig. 2. Detailed descriptions of SFC and GRAFCET in different versions can be found, among others, in [2, 5, 12, 17, 25–27, 30, 31].

5 ALU and PRALU

Relation between the languages ALU ("Algoritm Logicheskogo Upravleniya"—"Logical Control Algorithm") and PRALU is more or less the same as between GRAFCET and SFC: ALU is a language for specification of a structure of a parallel control algorithm, and PRALU is its concretization for the systems with binary signals [15].

An algorithm described in PRALU consists of the *chains*, which can be grouped into *sentences*. A chain consists of an *initial label*, a sequence of operations and a *final label*. The initial and final labels are the sets of natural numbers (a final label may be empty). If the initial labels have a non-empty common part, they must be equal (the sets of chains with the same initial labels are usually grouped into a sentence, then the initial label is written only once). The operations belong to one of two types: *operations of action* (preceded by symbol '-') and *operations of waiting* (preceded by symbol '-'), typically both represented by the elementary conjunctions. A waiting operation means that a chain waits until the conjunction obtains the value

1. An action operation assigns to the variables participating in the conjunction the values which turn the conjunction to 1.

Execution of the chains is controlled by a *firing set* N. Initially $N = \{1\}$. If an initial label μ of a chain is a subset of current firing set ($\mu \subseteq N$) and the condition expressed by the first waiting operation of the chain is satisfied (if the first operation of the chain is not a waiting operation, then the condition is understood as always satisfied), then the chain is activated. It means that the elements of μ are removed from N and the operations of the chain are executed sequentially, until the final label ν is attained. Then the chain is deactivated, and the elements of ν are added to the set N. So, the structure of a PRALU algorithm corresponds to structure of a binary Petri net, like an SFC program, but it is restricted to an EFC-net with single-token initial marking (an α-net). Another difference between SFC and PRALU is that in SFC the actions are associated with the places of an underlying Petri net, and in PRALU they are associated with the transitions.

A PRALU description can be hierarchical; then the actions initialising the sub-programs, also described in PRALU, are used.

More details about PRALU and implementation of the PRALU algorithms can be found in [15, 36–38, 41] and in the chapter "Circuit Implementation of Parallel Logical Control Algorithms Represented in PRALU Description". A simple example of a logical control algorithm specified by PRALU (taken from [38]) is shown below.

$$
\begin{array}{ll}
1: & -u \to ab -' u \to 2.3 \\
2: & -'vw \to' bc -' w \to b \to' c \to 2 \\
 & -v \to' ac \to 4.5 \\
3: & -uw \to d \to 6 \\
4: & -'u'v \to a - u \to' a \to 4 \\
 & -u \to a'b \to 7 \\
5: & -'vw \to c \to 8 \\
6.7.8: & \to' a'd -' w \to .
\end{array}
$$

Here the execution starts from the first chain, and the algorithm waits until $u = 1$. Then it assigns value 1 to the variables a and b and waits until the input variable u obtains value 0 ($'x$ means the negation of x). Then the first chain is deactivated, and the firing set obtains value $\{2, 3\}$, which is a condition of activation of the chains with the initial labels $\{2\}$ and $\{3\}$. From this point a possibility of parallel execution of the processes starts. A chain of the sentence with the initial label $\{2\}$ will be activated dependently on the values of the input variables v and w: if $v = 1$ then the second one is activated, if $v = 0$ then the first one will be activated when variable w turns to be 1. The chain with the initial label $\{3\}$ will be activated when $u = w = 1$, and so on.

6 Some Other Approaches

Below several other approaches to applying the Petri net models to logical control are mentioned.

- The control processes are often not purely binary but have to deal with the continuous signals. For this reason the *hybrid Petri nets* are used, having binary and continuous parts. Such models for the control systems are described, among others, in [17, 80, 81].
- The model named *reactive Petri nets* is presented in [60–62]. It is intended for specifying the reactive control systems. The model is a kind of coloured Petri nets, including inputs, outputs, time dependencies and priorities. The methods of FPGA implementation through translation to VHDL are designed for such nets.
- A format representing the control interpreted Petri nets in such a form which can serve both for formal verification using the model checking and for logical synthesis (in form of rapid prototyping for FPGA structures) is described in [24, 51, 82].
- In [43] a description of a parallel control system using the logical sequents was proposed. Basing on this approach, the *Petri Net Specification Format* (PNSF) was designed [47]. PNSF is a textual format which in its first version was intended for specification of a behavior described by an interpreted Petri net, with both Mealy and Moore outputs possible, to be implemented in a parallel controller. Its next version, PNSF2, was enhanced with hierarchy and possibilities of representing the coloured Petri nets [83]. The following version, PNSF3, is based on XML and intended first of all for simulation of the Petri net models of logic controllers [48].
- *Real-time coloured Petri nets* (RTCP-nets) is a model based on timed coloured Petri nets and intended for modelling and analysis of embedded real-time systems [84, 85]. Some ideas from this model were used in Alvis—a language for modelling and formal verification of concurrent systems [86].

7 Conclusions

There exists a mature theory of Petri nets in general and, in particular, there exists a deeply developed theory of the interpreted Petri nets, intended for specifying and modelling the logical control algorithms. The theory includes a wide range of Petri net-based models, methods of their verification and implementation. There is also a range of practically oriented Petri net-based languages for description of logical control algorithms, with Sequential Function Chart as the most popular among them, included in the international and European standards, and supported by the leading PLC manufacturers, such as Siemens and Schneider Electric. A lot of software tools exist, supporting the Petri net models and their analysis and implementation in the logic controllers.

On the other hand, the Petri net-based models and languages definitely do not belong to the popular ways of design of the control systems used in the engineering practice. Nevertheless, there are the "success stories" of using the Petri nets in design and verification of control processes, such as verification of control procedures of chemical plants in Germany [88], design of traffic control systems in Brasil [87], control of robots [89] and other industrial applications (some of them are described in [90–93]). It allows to hope that popularity of the Petri net-based approaches in logical control will be growing.

Acknowledgments The author is grateful to I. Grobelna and R. Wiśniewski for valuable comments which helped to improve this chapter.

References

1. Tsypkin, Ya Z. (1971). *Adaptation and learning in automatic systems*. New York: Academy Press.
2. Lewis, R. W. (1998). *Programming industrial control systems using IEC 1131–3*. IEE Control Engineering Series: IEE.
3. Liao, S., Tjiang, S., & Gupta, R. (1997). An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment, In *Proceedings of the 34th Design Automation Conference*, pp. 70–75.
4. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231–274.
5. Lepers, H. (2007). *SPS-Programmierung nach IEC 61131-3—Mit Beispielen für CoDeSys und Step7*. German: Franzis Verlag.
6. Baranov, S. (1994). *Logic synthesis for control automata*. Boston: Kluwer Academic Publishers.
7. Sklyarov, V., Skliarova, I., Barkalov, A., & Titarenko, L. (2014). *Synthesis and optimization of FPGA-based systems*. Lecture Notes in Electrical Engineering Cham: Springer International Publishing Switzerland.
8. Borowik, G., Rawski, M., Łabiak, G., Bukowiec, A., & Selvaraj, H. (2010). Efficient logic controller design, In *5th International Conference on Broadband and Biomedical Communications*, pp. 1–6, Malaga, Spain.
9. Lee, B., & Lee, E. A. (1998). Hierarchical Concurrent Finite State Machines in Ptolemy, In *Proceedings of the International Conference on Application of Concurrency to System Design*, pp. 34–40, Fukushima, Japan.
10. Sklyarov, V. (1983). Finite state machines with stack memory and their automatic design. *In Proceedings of USSR conference on computer-aided design of computers and systems, Part2*, 66–67 (in Russian).
11. Sklyarov, V., & Skliarova, I. (2013). Hardware implementations of software programs based on hierarchical finite state machine models. *Computers and electrical engineering*, *39*, 2145–2160.
12. Adamski, M., & Chodań, M. (2000). Modelling of Discrete Control Devices Using SFC. Wydawnictwo Politechniki Zielonogórskiej, Zielona Góra (in Polish).
13. T. Murata (1989) Petri nets: Properties, analysis and applications, *Proceedings of the IEEE*, *77*(4).
14. Girault, C., & Valk, R. (2001). *Petri Nets for system engineering: A guide to modeling, verification, and applications*. New York: Springer.
15. Zakrevskij, A., Pottosin, Yu., & Cheremisinova, L. (2009). *Design of logical control devices*. Tallinn: TUT Press.

16. Petri, C. A. (1962). *Kommunikation mit Automaten*, Ph.D thesis, Schriften des IIM nr 3, Institut für Instrumentelle Mathematik, Bonn, Germany.
17. David, R., & Alla, H. (2010). *Discrete, continuous, and hybrid Petri Nets*. Berlin: Springer.
18. Daclin, E., & Blanchard, M. (1976). *Synthese des Systemes Logiques*, Cepadues.
19. Silva, M., & David, R. (1977). On the programming of asynchronous sequential systems by logic equations, *IFAC Int. Symp. on Discrete Systems*, pp. 52–62.
20. Silva, M., & Teruel, M. (1998). DEDS along their life-cycle: Interpreted extensions of Petri Nets, *IEEE International Conference on Systems, Man and Cybernetics*.
21. Silva, M. (2013). Half a century after Carl Adam Petri's Ph.D. thesis: A perspective on the field. *Annual Reviews in Control*, *37*(2), 191–219.
22. Andrzejewski, G. (2003) *Program Model of Interpreted Petri Net for Design of Digital Microsystems*, PhD thesis, Prace Naukowe z Automatyki i Informatyki, T. 2, Uniwersytet Zielonogorski, Zielona Góra (in Polish).
23. Andrzejewski, G., & Karatkevich, A. (2003). Interpreted Petri nets in system design, *Sbornik trudov X mezdunarodnoj naucno-techniceskoj konferencii Masinostroenie i technosfera XXI veka (Machine-building and technosphere of the XXI century)*, Donetsk, Ukraine, pp 7–10.
24. Grobelna, I., & Adamski, M. (2011). Model checking of Control Interpreted Petri Nets, *Proceedings of the 18th International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES)*, pp. 621–626.
25. David, R., & Alla, H. (1992). *Petri Nets and Grafcet: Tools for modelling discrete event systems*. New York: Prentice Hall.
26. Le Grafcet, ontil de representation du cahier de charges d'un automatisme logique, *Rapport final de la Commission AFCET*, Paris (1977).
27. Le Grafcet, diagramme functionnel des automatismes sequentels, *Rapprot de la Commission ADEPA sur la normalisation du GRAFCET*, Montrouge (1977).
28. *IEC 61131-3, International Standard, Programmable controllers—Part 3: Programming languages*, Edition 3.0, International Electrotechnical Commission (2013).
29. Zajac, W., Kołopien'czyk, M., & Andrzejewski, G. (2014). Modelling and Synthesis of Parallel Traffic Control Algorithms with Time Dependencies, *New trends in digital system design*, Fortschr.-Ber. VDI Reihe 10 Nr. 836, VDI Verlag, Düsseldorf, pp. 94–109.
30. *STEP 7 Professional V13.0 System Manual*, Siemens AG Industry Sector, Nürnberg, Germany (2004).
31. ABB Group, Automation Builder 1.1—Complete English Documentation (2015). (available online at www.abb.com)
32. Yuditski, S. A., Tagayevskaya, A. A., & Yefremova, G. K. (1977). *A Language for Algorithmic Design of Discrete Control Devices*, preprint of Institute of Control Sciences, Moscow (in Russian).
33. Zakrevskij, A. D. (1981). A-net—a functionsl model of a discrete system, *Doklady AN BSSR*, vol. 22, Nr. 1, pp. 714–717 (in Russian).
34. Zakrevskij, A. D. (1984). Parallel automaton, *Doklady AN BSSR*, vol. 28, Nr. 8, pp. 717–719 (in Russian).
35. Zakrevskiy, A. D. (1986). Petri nets modeling of logical control algorithm. *Automatic Control and Computer Sciences*, *20*(6), 38–45.
36. Zakrevskii, A. D. (1987). The analysis of concurrent logic control algorithms. *Fundamentals in computational theory* (pp. 497–500). Lecture Notes in Computer Science Berlin: Springer.
37. Zakrevskij, A. D. (1989). To the theory of parallel algorithms of logical control. *Izvestiya AN SSSR, Tekhnicheskaya Kibernetika, Nr. 5*, 179–191 (in Russian).
38. Zakrevskij, A. D. (1999). *Parallel logical control algorithms*, Institute of Engineering Cybernetisc of the National Academy of Science of Belarus (in Russian).
39. Steinbach, B., & Zakrevskij, A. D. (2000). Parallel automaton—basic model, properties and high-level diagnostics, *Proceedings of the 4th International Workshop on Boolean Problems*, Freiberg, pp. 151–158.
40. Bibilo, P. N. (2000). *VHDL fundamentals*. Moscow: Solon-R (in Russian).

41. Cheremisinova, L. (2002). Realization of Parallel Logical Control Algorithms. *Institute of Engineering Cybernetisc of the National Academy of Science of Belarus* (in Russian).
42. Adamski, M. (1981). (1981). Realization of Petri Nets using PLA, *Krajowa Konf. Teoria Obwodow i Uklady Elektroniczne*, Drzonkow, pp. 455–459 (in Polish).
43. Adamski, M. (1990). *Design of digital devices by systematic structural method*. Zielona Góra: Wyzsza Szkola Inzynierska (in Polish).
44. Adamski, M. (1991). Parallel controller implementation using standard PLD software, *FPGAs: International Workshop on Filed Programmable Logic and Applications*, Abingdon EE&SC Books, pp. 296–304.
45. Adamski, M. (2005). Design of Embedded Control Systems. *Formal logic design of reprogrammable controllers*. New York: Springer.
46. Bubacz, P., & Adamski, M. (2006). Heuristic algorithm for an effective state encoding for reconfigurable matrix-based logic controller design, *Programmable Devices and Embedded Systems—PDeS 2006: proceedings of IFAC workshop*, Brno, pp. 236–241.
47. Kozłowski, T., Dagless, E. L., Saul, J. M., Adamski, M., & Szajna, J. (1995). Parallel controller synthesis using Petri nets. *IEE Proceedings, Computers and Digital Techniques*, *142*(4), 263–271.
48. Węgrzyn, A., & Węgrzyn, M. (2005). Design of Embedded Control Systems. *A new approach to simulation of concurrent controllers*. New York: Springer.
49. Zakrevskij, A., Karatkevich, A., & Adamski, M. (2002). A method of analysis of operational Petri nets. *Advanced computer systems: Eight international conference, ACS 2001* (pp. 449–460). Boston: Kluwer Academic Publishers.
50. Karatkevich, A. (2007). *Dynamic analysis of Petri net-based discrete systems*, Lecture Notes in Control and Information Sciences, Berlin: Springer.
51. Grobelna, I., Wiśniewska, M., Wiśniewski, R., Grobelny, M., & Mróz, P. (2014). Decomposition, validation and documentation of control process specification in form of a Petri net, *7th International Conference on Human System Interactions (HSI)*, Lisbon, Portugal, pp. 232–237.
52. Węgrzyn, A., (2003). *Symbolic Analysis of Binary Control Devices Using Selected Methods of Analysis of Petri Nets*, Ph D thesis, Prace Naukowe z Automatyki i Informatyki, T. 3, Uniwersytet Zielonogórski, Zielona Góra (in Polish).
53. Węgrzyn, M. (2010). Modelling of Petri nets in VHDL. *Electrical Review, Nr.*, *1*, 212–216 (in Polish).
54. Wiśniewska, M. (2012). *Application of hypergraphs in decomposition of discrete systems*, PhD thesis, Lecture Notes in Control and Computer Science, Vol. 23, University of Zielona Góra, Zielona Góra.
55. Węgrzyn, M., Adamski, M., Karatkevich, A., & Munoz, A. (2014). FPGA-based embedded logic controllers, *7th International Conference on Human System Interactions (HSI)*, Lisbon, Portugal, pp. 249–254.
56. Wiśniewski, R., Stefanowicz, Ł., Bukowiec, A., & Lipiński, J. (2014). Theoretical aspects of Petri nets decomposition based on invariants and hypergraphs, 8th International Conference of Multimedia and Ubiquitous Engineering (MUE). *Zhangjiajie, China, Lecture Notes in Electrical Engineering*, *308*, 371–376.
57. Łabiak, G. (1999). *Modelling statechart diagrams by means of Petri nets, Advanced Computer Systems—ACS '99: Sixth Inernational Conference: Proceedings* (pp. 253–259). Poland: Szczecin.
58. Miczulski, P. (2005). Design of Embedded Control Systems. *Calculating state spaces of hierarchical Petri nets using BDD*. New York: Springer.
59. Łabiak, G., & Karatkevich, A. (2014). The use of algebraic decision diagrams for algebraic analysis of n-bounded Petri Nets, *New trends in digital systems design*, Fortschritt—Berichte VDI : Nr. 836, Düsseldorf, pp. 56–67.
60. Yakovlev, A., Gomes, L., & Lavagno, L. (2000). *Hardware design and Petri nets*. Berlin: Springer.
61. Gomes, L., Barros, J. P., & Costa, A. (2005). Design of Embedded Control Systems. *Structuring mechanisms in Petri net models*. New York: Springer.

62. Gomes, L., Costa, A., Barros, J. P., & Lima, P. (2007). From Petri net models to VHDL implementation of digital controllers, *33rd Annual Conference of the IEEE Industrial Electronics Society IECON 2007*, pp. 94–99.
63. Valette, R., Courvoisier, M., Begou, J. M., & Albukerque, J. (1983). Petri net based programmable logic controllers, *Proceedings of 1st IFIP Conference: Computer Application in Production and Engineering*, pp. 103–116.
64. Ferrarini, L. (1992). An incremental approach to logic controller design with Petri nets. *IEEE Transactions on System, Man and Cybernetics*, *22*(3), 461–474.
65. Biliński, K., Adamski, M., Saul, J. M., & Dagless, E. L. (1994). Parallel Controller Synthesis from a Petri Net Specification, *Proceedings of the conference on European design automation EURO-DAC '94*, pp. 96–101.
66. K. Biliński, *Application of Petri Nets in Parallel Controller Design*, Ph.D. Thesis, University of Bristol, 1996.
67. Park, E., Tilbury, D. M., & Khargonekar, P. P. (2001). A modeling and analysis methodology for modular logic controllers of machining systems using Petri net formalism. *IEEE Transactions on System, Man, and Cybernetics - Part C: Applications and Reviews*, *31*(2), 168–188.
68. Marranghello, N., de Oliviera, W., & Damiani, F. (2004). A Petri net based timing model for hardware/software co-design of digital systems, *IEEE Asia-Pacific Conference on Circuits and Systems*, Tainan, pp. 65–68.
69. Zakrevskij, A. (1986). Elements of the theory of α-nets, *Design of logical control systems*, Institute of Engineering Cybernetisc of the Academy of Sci. of BSSR, pp. 4–12 (in Russian).
70. Pottosin, Yu. (2005). *Optimal State Assignment of Synchronous Parallel Automata* (pp. 111–124). Springer, New York: Design of Embedded Control Systems.
71. Cheremisinova, L. (2005). *Optimal State Assignment of Asynchronous Parallel Automata* (pp. 139–149). Springer, New York: Design of Embedded Control Systems.
72. Andrzejewski, G. (2005). *Hierarchical Petri Nets for Digital Controller Design*. Springer, New York: Design of Embedded Control Systems.
73. Karatkevich, A., & Andrzejewski, G. (2006). Hierarchical Decomposition of Petri Nets for Digital Microsystems Design, *International Conference on Modern Problems of Radio Engineering, Telecommunications, and Computer Science—TCSET 2006*, IEEE, Lviv, pp. 518–521.
74. Karatkevich, A. (2008). On macroplaces in Petri nets, *2008 East-West Design & Test Symposium (EWDTS)* , IEEE, Lviv, pp. 418–422.
75. Pais, R., Gomes, L., & Barros, J. P. (2011). From UML state machines to Petri nets: History attribute translation strategies, *IECON 2011—37th Annual Conference on IEEE Industrial Electronics Society*, IEEE, Melbourne, pp. 3776–3781.
76. Best, E., & Coutny, M. (1992). Petri net semantics of priority systems. *Theoretical Computer Science, Nr.96*, 175–215.
77. Łabiak, G. (2010). Transition orthogonality in statechart diagrams and inconsistencies in binary control system. *Przeglad Elektrotechniczny*, *86*(9), 130–133.
78. J. L. Peterson, *Petri net theory and the modeling of systems*, Prentice-Hall, 1981
79. Popova, L. (1991). On Time Petri Nets. *J. Inform. Process. Cybern.*, *27*(4), 227–244.
80. Pais, H., David, R., & Le Bail, J. (1991). Hybrid Petri nets, *Proceedings of the European Control Conference*, Grenoble.
81. Hummel, Th, & Fengler, W. (2005). *Design of Embedded Control Systems Using Hybrid Petri Nets* (pp. 139–149). Springer, New York: Design of Embedded Control Systems.
82. Grobelna, I. (2011). Formal verification of embedded logic controller specification with computer deduction in temporal logic. *Przeglad Elektrotechniczny*, *87*(12A), 47–50.
83. Węgrzyn, A., & Węgrzyn, M. (2006). Selected Textual Formats of Petri Net Specification Describing Control Algorithms, *Pomiary, Automatyka Kontrola*, Nr. 6bis, pp. 29–31 (in Polish).
84. Szpyrka, M. (2006). Analysis of RTCP-nets with Reachability Graphs. *Fundamenta Informaticae*, *74*(2–3), 375–390.
85. Szpyrka, M. (2007). Analysis of VME-Bus communication protocol - RTCP-net approach. *Real-Time Systems*, *35*(1), 91–108.

86. Szpyrka, M., Matyasik, P., Mrówka, R., & Kotulski, L. (2014). Formal Description of Alvis Language with α^0 System Layer, *Fundamenta Informaticae*, Vol. 129 Nr. 1–2, January.
87. Perkusich, A., de Araujo, L. M., de Coelho, R. S., Gorgonio, K. C., & Lemos, A. J. P. (1999). Design and Animation of Colored Petri Nets Models for Traffic Signals, *Proceedings of the 2nd Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, Aarhus, pp. 99–118.
88. Genrich, H. J., Hanisch, H.-M., & Wöllhaf, K. (1994). Verification of Recipe-Based Control Procedures by Means of Predicate/Transition Nets, *Proceedings of the 15th International Petri Net Conference on Application and Theory of Petri Nets*, Zaragoza, *Lecture Notes in Computer Science*, Vol. 815, Springer-Verlag, pp. 278–297.
89. Caccia, M., Coletta, P., Bruzzone, G., & Veruggio, G. (2005). Execution control of robotic tasks: a Petri net-based approach. *Control Engineering Practice*, *13*(8), 959–971.
90. DiCesare, F., Harhalakis, G., Proth, J. M., Silva, M., & Vernadat, F. B. (1993). *Practice of Petri Nets in Manufacturing*, Chapman and Hall.
91. Giua, A., & DiCesare, F. (1993). *GRAFCET and Petri Nets in Manufacturing, Intelligent Manufacturing: Programming Environments for CIM* Advanced Manufacturing Series, (pp. 153–176). London: Springer.
92. Desrochers, A. A., & Al'Jaar, R. Y. (1995). *Applications of Petri nets in Manufacturing Systems: Modelling*. Control and Performance Analysis: IEEE Press.
93. Pawlewski, P. (Ed.). (2012). *Petri Nets*. InTech: Manufacturing and Computer Science.

Synthesis and Implementation of Parallel Logic Controllers in All Programmable Systems-on-Chip

Valery Sklyarov, Iouliia Skliarova and João Silva

Abstract The chapter is dedicated to the design of logic controllers with customizable behavior in all programmable systems-on-chip in such a way that the desired functionality is defined in software of a processing system and realized in hardware of reconfigurable logic. The controllers implement algorithms described in form of parallel hierarchical graph-schemes that are built in software from predefined modules. Parallel hierarchical circuits of the controllers are mapped to the reconfigurable logic customized from software through high-performance interfaces. The circuits generate control signals to determine the desired functionality of external devices. A number of experiments are done in Xilinx Zynq-7000 microchips and the results are reported.

Keywords Hardware/software architectures · Parallel logic controllers · Hierarchical finite state machines · Hierarchical algorithms · Hardware/software interactions

1 Introduction

Nowadays, the development of software and hardware becomes more and more interrelated [1]. The emphasis has significantly shifted from general-purpose to application-specific products in the form of embedded processing modules in various areas such as communications, industrial automation, automotive computers, and home electronics. There is a tendency to integrate components on a chip that not so long ago were separated and implemented as autonomous devices. For example, the Zynq-7000 [2] all programmable system-on-chip (APSoC) incorporates a processing

V. Sklyarov (✉) · I. Skliarova · J. Silva
Department of Electronics, Telecommunications and Informatics/IEETA,
University of Aveiro, Aveiro, Portugal
e-mail: skl@ua.pt

I. Skliarova
e-mail: iouliia@ua.pt

J. Silva
e-mail: jpss@ua.pt

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_2

system (PS) that combines the industry-standard ARM dual-core Cortex^{TM}-A9 RISC processor and a number of peripherals such as memory controllers, USB, Gigabit Ethernet, and UART. The same micro-chip contains a built-in gate array (programmable logic—PL) from the Artix-7 or Kintex-7 FPGA families that is linked with the PS through on-chip interfaces.

APSoCs like Zynq [2] can run software that interacts with parallel processing elements (PE) mapped to hardware. The main objective of any PE is to provide greater performance than an equivalent software component with similar functionality that is typically composed of a set of functions in C or methods in Java. A parallel logic controller can be seen as one of application-specific PEs that gets inputs from the controlled systems and generates outputs that ensure the desired functionality. Real-time systems may require high-speed control that can be provided more easily in hardware rather than in software. Besides, control circuits are often used in such hardware components that replace software functions [3].

For many practical applications (such as knowledge-based systems in [4]) interaction between programmable logic controllers and software in a PC is widely used. We suggest in this chapter to provide better support for such interactions using APSoCs that run software in the dual-core processing system and hardware in the programmable logic. The emphasis is done on the following issues:

1. Support for modularity, hierarchy and parallelism in hardware (in the PL of APSoC) based on hierarchical (HFSM) and communicating (CFSM) finite state machines [5] with such functionality that can be customized and modified from software of APSoC running in the PS.
2. Interactions between a programmable parallel logic controller implemented in the PL and software in the PS through interrupts, general-purpose and high-performance ports.
3. Dynamic reconfiguration of the controller from software of the PS based on the methods [6] and potentially applying the knowledge-based technique from [4].

The remainder of the chapter is organized in five sections. Section 2 suggests architectures of parallel logic controllers implemented in APSoCs and the methods of interaction between hardware and software components. Section 3 describes the design and implementation of parallel logic controllers with dynamically modifiable functionality providing support for modularity and hierarchy. Section 4 gives more details about hardware/software interactions. Section 5 discusses the details of implementations and examples. Section 6 concludes the chapter.

2 The Proposed Software/Hardware Architecture

Figure 1 shows the proposed hardware/software architecture. A reconfigurable parallel logic controller is implemented in the PL and we will consider below the following two models for such controllers: parallel hierarchical finite state machines (PHFSMs) [3] and communicating finite state machines (CFSMs) [5].

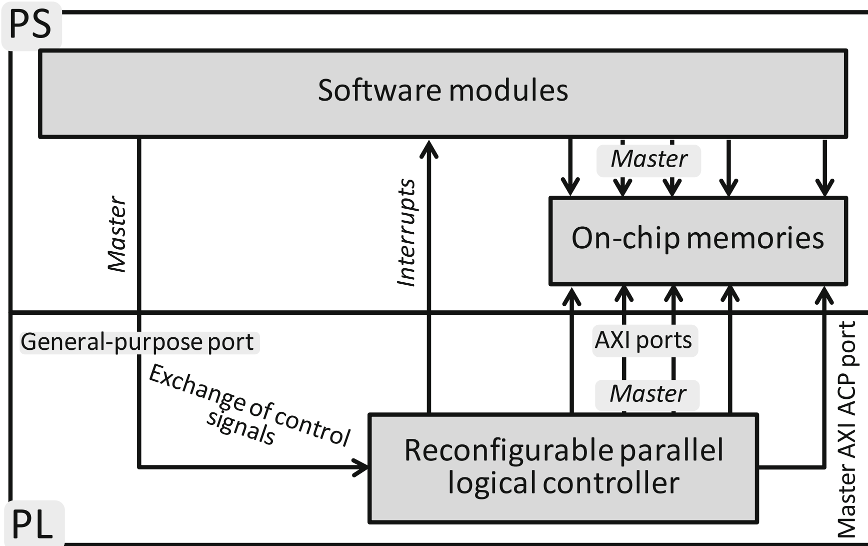


Fig. 1 Hardware/software architecture

Software modules in the PS are responsible for the following three functions:

1. Higher-level control that enables lower-level modules of PHFSM/CFSM to be managed. This means that the modules are not hard linked in the PL and can be activated/deactivated from software which much like [4] may use knowledge-based technique.
2. Run-time reconfiguration of lower-level modules allowing different functionalities to be implemented using the same hardware.
3. Test and debug of the lower-level modules.

Interaction between software and hardware modules is provided through the following interfaces:

1. General-purpose ports (GPP) [2] for exchange of control signals.
2. High-performance ports (HPP) to configure (reconfigure) modules of the parallel logic controller.
3. Interrupts generated in hardware and handled by software to support high-priority requests from hardware that need immediate reaction, which is important for real-time systems.

Figure 2 shows communication mechanisms between software and hardware with more details. The PHFSM/CFSM contains modules that can be executed in parallel. Any module is considered to be either a conventional finite state machine (FSM) or a hierarchical FSM (HFSM) and has pre-defined signals that are:

a) An input signal *start* indicating that the module has to be reset to the initial state and begin execution.

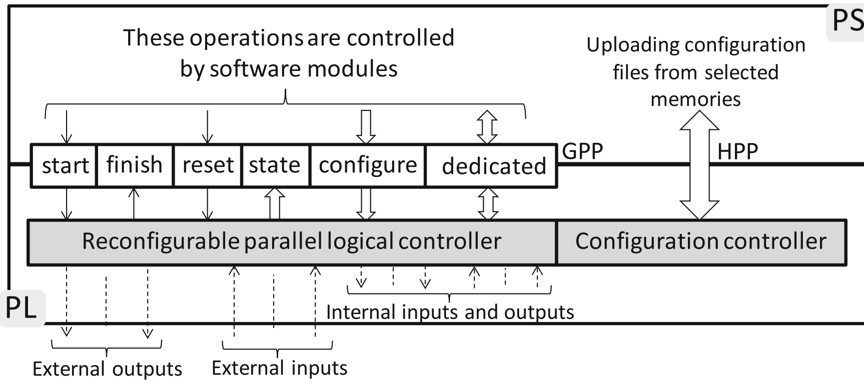


Fig. 2 Details of interactions between software and hardware modules

b) An output signal *finish* designating that the module has completed the associated operations and is suspended.
c) An input signal *reset* requiring transition to the initial state of the module. This signal may also reset the relevant registers in the attached execution unit (datapath).
d) An output vector named *state* represents the current state of HFSM/FSM memory (state register). This vector can be used efficiently for debugging purposes. Indeed, software is capable of monitoring this signal and concluding if the desired functionality is properly provided or if there is an unusual situation. Many potential deadlocks can be found and eliminated.
e) An input signal *configure* requests customization of the module and points to the first address in on-chip memory with the reconfiguration file. On such a request the module is reconfigured by a configuration controller and as soon as this operation is completed the signal *finish* is generated.
f) Some signals are dedicated to particular module functionality and we will discuss them later.

Software modules set/check the GPP signals using two ways:

a) Periodically and on internal requests generated according to the implemented algorithms. For example, as soon as one task is completely solved the hardware module responsible for the task may be reconfigured to solve the subsequent task.
b) Immediately on interrupts from hardware modules.

Hardware modules may be configured statically or dynamically. Static configuration is done when the relevant bit-stream is uploaded to the PL section. Dynamic reconfiguration is provided during execution time, i.e. after bit-stream has been loaded. This is done with the aid of the methods described in [6] (see the next section). PHFSM/CFSM may be used for the following three types of applications:

1. External devices connected to APSoC pins, such as those described in [7].
2. Internal blocks that may be used for different purposes, for example to accelerate time consuming segments of software modules.
3. A composition of external and internal devices, for example, some modules of the HFSM/CFSM may control components of an assembly line [7] and some other modules may be used for solving optimization problems such as planning the sequence of operations, etc.

Apart from applications described above, PHFSMs/CFSMs can be used as hardware accelerators of software programs, such as [1]. We will show below that for such applications capabilities of parallelism, modularity, hierarchy and dynamic reconfiguration are also very useful and important.

3 Design and Implementation of the Parallel Logic Controller

We have already mentioned that the parallel logic controllers considered here are based on different FSM models. Basically, we can distinguish three types of FSM models, which are *simple sequential*, *hierarchical*, and *parallel*. In turn, they can be further divided (for example, we can consider recursive and iterative hierarchical models).

Methods of synthesis for *simple sequential* FSMs are very well studied [8, 9] and they are considered just as a basis for more complicated hierarchical and parallel FSMs.

A *hierarchical* FSM is composed of other hierarchical and simple sequential FSMs (modules), which can be activated much like procedures in software programs. Thus, any module can be triggered from either another or the same module (see Fig. 3) [5].

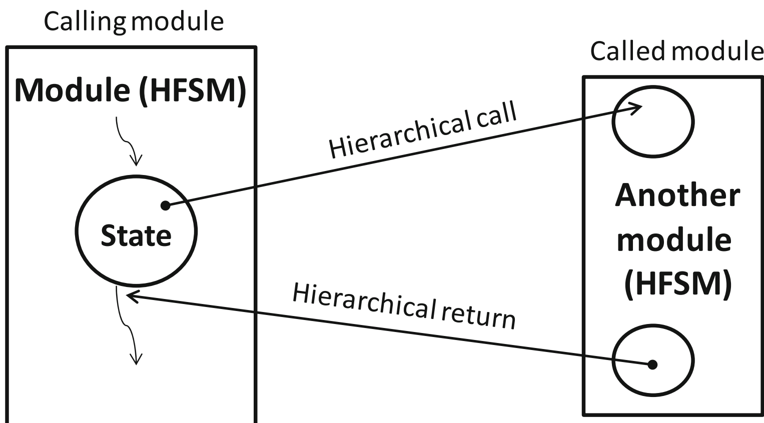


Fig. 3 Execution of hierarchical modules

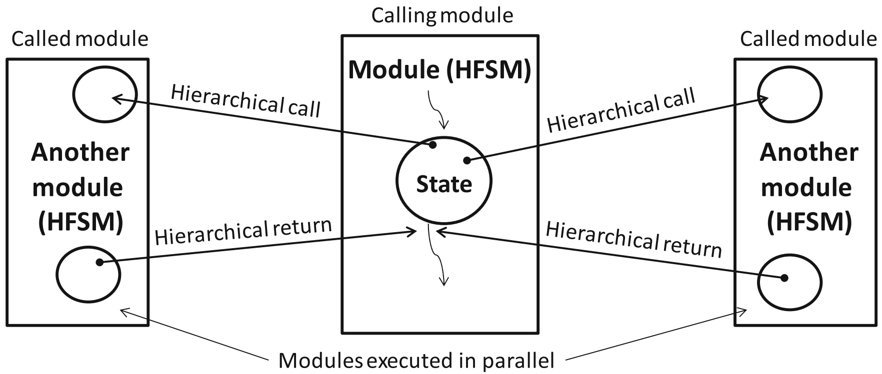


Fig. 4 Execution of parallel modules

A *parallel* FSM enables different modules to be executed in parallel (see Fig. 4). Note that generally any electronic device deals with simultaneous processing of analog/digital signals. Thus, it is parallel by definition. However, circuit level of parallelism does not give answers to many questions appearing at the algorithmic level of specification. For example, how can different branches of algorithms be executed in parallel, how can pipelining technique be applied, etc. In [5] all necessary answers to such questions are given.

The most interesting approach is a combination of parallel and hierarchical capabilities within the same FSM, which becomes a PHFSM.

Reconfiguration of HFSMs/CFSMs can be done with the aid of the methods [6] which permit HFSM/CFSM circuits to be built from reloadable memories that determine the desired functionality. The memories (that are embedded or distributed PL blocks) can be updated at execution time and thus the operations of the HFSMs/CFSMs can be changed in accordance with the requirements that might depend on some factors [3, 4].

Since HFSMs/CFSMs are composed of modules that may be replaced if required, different control algorithms specified by the modules can be selected during execution time in order to adjust parameters of the controlled devices. Thus, we can apply the strategy "try, test and replace if required". Besides, any module can be updated with an improved version without modification of surrounding modules [3]. For example, the PS evaluates the functionality of the controlled devices and verifies if the established requirements are satisfied. If based on the result of evaluation the PS makes a conclusion that some modes or algorithms applied to the controlled devices may be improved then the set of active modules implemented in the PL can be updated and some of such modules may be reconfigured using the methods [6].

Hierarchy and parallelism can be described using various methods such as [3, 10–12]. We will use parallel hierarchical graph-schemes (PHGSs) [5]. An example of a PHGS which describes functionality of a self-controlled transport section from [13] is given in Fig. 5. The algorithm is composed of 7 modules $Z_0, \ldots, Z_6$. Some

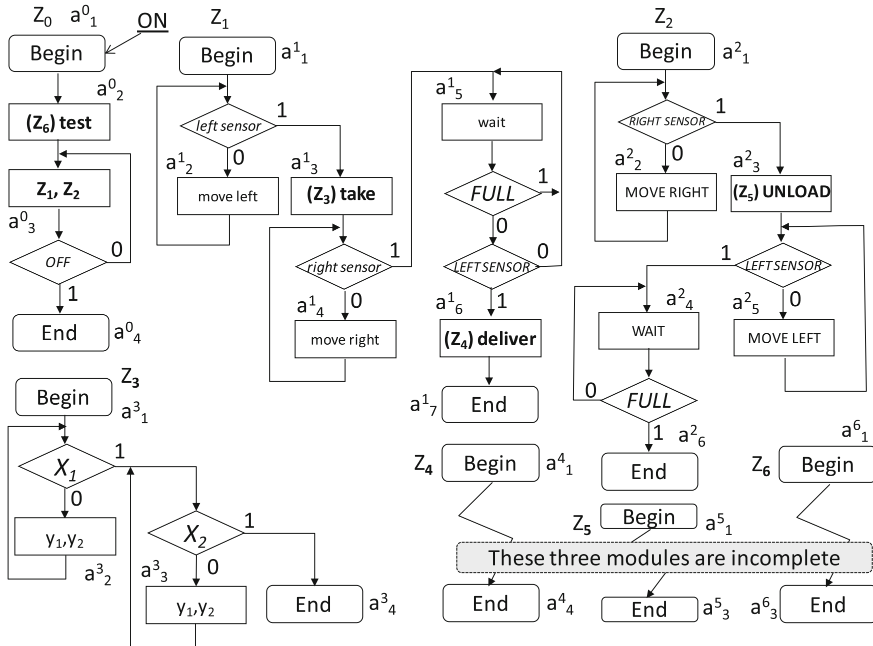


Fig. 5 An example of parallel hierarchical algorithms for a logic controller from [13]

of the modules, namely $Z_1, \ldots, Z_6$, are activated hierarchically and some of them, namely Z_1, Z_2, are called in parallel. Labels like a^0_1 and a^0_2 represent states [5]. Rhomboidal nodes contain logical conditions that are formed by sensors of the logic controller and enable the sequence of execution of the algorithm to be properly selected. For example, if $OFF = 0$ in the node a^0_3 of the module Z_0 the execution of the rectangular node a^0_3 is repeated. If $OFF = 1$ in the node a^0_3 the module Z_0 is terminated. Microoperations (like y_1, y_2, *move left*, etc.) affect actuators of the controlled device forcing the required operations to be executed.

The modules can be activated from each other in such a way that:

a) the calling module is suspended;
b) the called module is executed;
c) as soon as the called module is terminated, the control has to be returned back to the calling module, i.e. the calling module continues its execution starting from a node following the node with the terminated called module. For example, the node a^0_2 of the calling module Z_0 activates the called module Z_6. After Z_6 is terminated, the control has to be returned back to Z_0 and the node a^0_3 has to be activated.

If two or more modules are activated in the same rectangular node they have to be executed in parallel. For example, the modules Z_1 and Z_2 have to be activated in parallel from the module Z_0. If two or more modules (the called modules) are called

in parallel from the calling module, the calling module is allowed to continue its execution if and only if all the called parallel modules have been completed. In other words if any of the called parallel modules is still functioning, the calling module has to be suspended. PHFSMs can formally be synthesized from PHGSs using the methods [3, 5, 13].

PHFSMs/HFSMs/FSMs may be connected in a network in such a way that they communicate with each other [5]. The communications we consider here are managed by software modules (see Fig. 2) in such a way that:

- Any FSM module can be *activated/reset/configured/tested* by software modules through GPPs and HPPs (see Fig. 2). Thus, many communication mechanisms in CFSMs [5] are provided by software.
- For such FSM states where some operations have to be immediately executed special interrupts from hardware to software are generated.
- Software modules check states of FSM modules and the interrupts from the FSM modules and make conclusion about subsequent operations.

4 Hardware/Software Interactions

Hardware/software interactions are supported by two hardware components that have been developed in the Vivado 2014.2 design suite for Zynq microchips. The first component GP_control provides support for interactions through GPPs and the second one, HP_control, enables dynamic reconfiguration to be done. Three Xilinx libraries proc_common, axi_lite_ipif, and axi_master_burst were used.

Data exchange through GPPs is provided through the PL registers mapped to an address range defined by the constant of Xilinx type SLV64_ARRAY_TYPE [14]. Interaction is organized through Xilinx modules in packages axi_lite_ipif and proc_common. From the side of hardware the constants C_ARD_ADDR_RANGE_ARRAY of Xilinx type SLV64_ARRAY_TYPE and C_ARD_NUM_CE_ARRAY of type INTEGER_ARRAY_TYPE have been properly customized selecting the required chip select and chip enable signals (many examples are given in [15]). The minimum allowed size of a memory segment is 1000_{16} (it is defined by the Xilinx constant C_S_AXI_MIN_SIZE) and it is almost always sufficient for all modules interacting with software in a way shown in Fig. 2. In rare cases when larger number of signals for GPPs is needed this constant can easily be increased (see details in [14]). Signals between the PS and the PL are transferred through registers in the PL addressed by the values in the constants and managed by the PS (the PS is the master and the PL is the slave). Hardware and software can be developed independently of each other using the defined transfer area to communicate. All projects for experiments were implemented as standalone. Other types of projects (such as running under Linux) can be prepared using the methods described in [15].

Reconfiguration of different FSM modules is done through HPPs and this requires the following customization:

1. The used memory (on-chip memory—OCM, or cache for our projects) was enabled and the size of transferred data (32 or 64 bits) was indicated.
2. The initial memory address needs to be chosen identically in software and in hardware. Software modules were developed in C language.

As soon as a request for configuration is set from software, the configuration controller in the PL copies data (allowing the chosen FSM module to be customized [6]) to the necessary memory blocks that are either embedded to the PL or distributed elements built from the PL look-up tables. It is done similarly to [16].

Reconfiguration data are kept in either OCM or cache filled in from a host PC. Copying data from the host PC to on-chip memories is done with the aid of projects from [15]. The memories are always considered to be slaves and the PS that copies data from the PC to the memories and the configuration controller in the PL that reads data and customizes the chosen FSM module are masters operating in different time slots. Configuration data are transferred from the PS to the PL in a burst mode as shown in Fig. 6.

The top module instantiates several components, two of which are *GP_control* and *Configuration module*. The remaining components are Xilinx intellectual property (IP) cores. The component *Burst reader* executes burst read (supported by the Xilinx component axi_master_burst [17]) and generates the signal *finished* as soon as reading is completed. After that HFSM/FSM memory blocks are loaded much like it is done in [16].

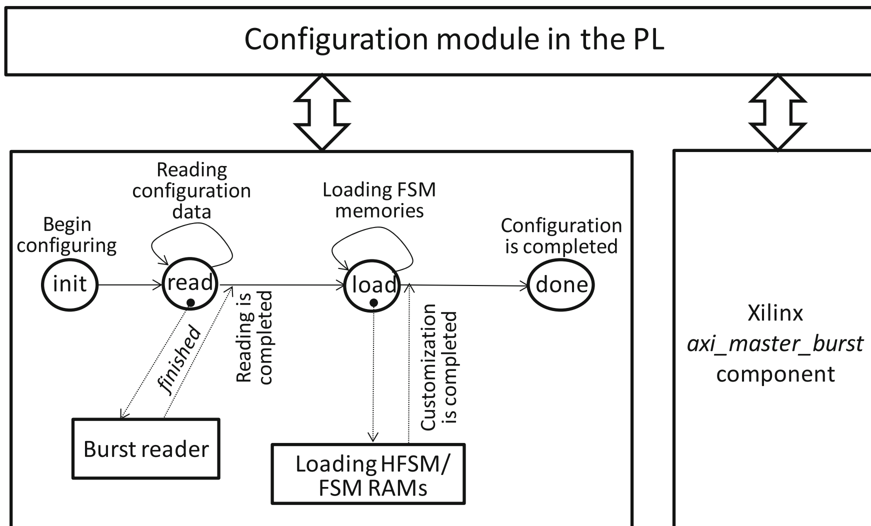


Fig. 6 Component diagram for configuration of FSM modules in burst mode

The sequence of operations *init*, *read*, *load*, and *done* is formed by a dedicated (not reconfigurable) HFSM module with the relevant states, two of which (*read* and *done*) involve hierarchical operations. The first operation is implemented in the *burst reader* and it is given in [15]. The second operation enables to load HFSM/FSM memory blocks that permit the desired customization of HFSM/FSM modules to be done.

Interrupts can be generated in any FSM module if immediate reaction is needed from the software modules. Interrupts are initiated by dedicated signals in some chosen HFSM/FSM states and processed in software by the interrupt handler. Many examples that demonstrate how interrupts can be processed in Zynq microchips are given in [15]. A similar technique is used in logic controllers that are considered here.

5 Implementations and Examples

Figure 7 shows the organization of the experiments. We used a multi-level computing system [18]. Configuration data are prepared in software of the host PC and saved in files that are copied to APSoC memories using projects from [15]. Modules of parallel logic controllers are created in the PL and managed from software of the PS. The latter and software of the host PC may also be responsible for verifying functionality of different HFSM/FSM modules. Standalone applications have been created and uploaded to the PS from Xilinx Software Development Kit (SDK) using methods described in [15]. Interaction is done through the SDK console window. All experiments were done in two Zynq-based prototyping systems: ZyBo [19] and ZedBoard [20]. Two examples are discussed in the subsequent sections.

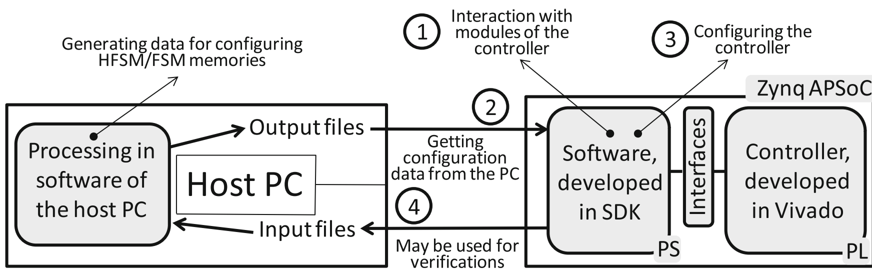


Fig. 7 Experimental setup

5.1 An Example of PHFSM-Based Hardware Accelerator

Let us consider a project demonstrating the use of PHFSM to accelerate computation of the greatest common divisor for *N* unsigned integers, where *N* is chosen to be 8. The intended functionality is demonstrated on an example of the following C function gcd with 8 arguments:

```
unsigned int gcd (unsigned int A, unsigned int B,
    unsigned int C, unsigned int D, unsigned int E,
    unsigned int F, unsigned int G, unsigned int H)
{
    return gcd(gcd(gcd(A,B), gcd(C,D)), gcd(gcd(E,F), gcd
        (G,H)));
}
```

This function permits the greatest common divisor of 8 operands A, B, C, D, E, F, G, and H to be found and calls another function gcd with two operands:

```
unsigned int gcd (unsigned int A, unsigned int B)
{
    unsigned int tmp;
    while (B > 0)
    {
        if (B > A)
        {
            tmp = A;
            A = B;
            B = tmp;
        }
        else
        {
            tmp = B;
            B = A % B;
            A = tmp;
        }
    }
    return A;
}
```

Clearly, four functions gcd(A, B), gcd(C, D), gcd(E, F), gcd(G, H) can be executed in parallel at the first step giving the results Result_A_B, Result_C_D, Result_E_F, and Result_G_H. At the second step, these results will be used as arguments for the functions: gcd(Result_A_B, Result_C_D), and gcd(Result_E_F, Result_G_H), which can also be executed in parallel giving the results Result_A_B_C_D, and Result_E_F_G_H. At the next (last) step the function gcd(Result_A_B_C_D, Result_E_F_G_H) computes the final greatest common divisor of 8 unsigned integers A, B, C, D, E, F, G, and H. All the above functions will be implemented in the PHFSM

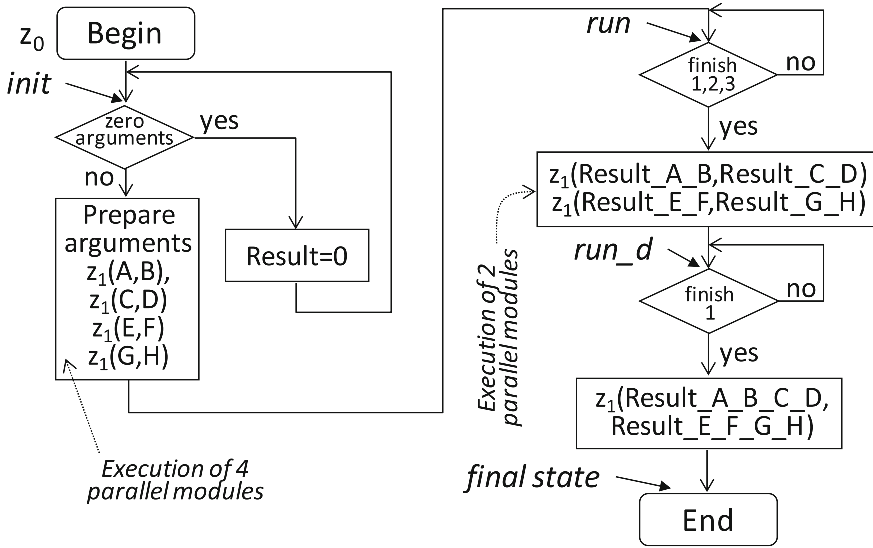


Fig. 8 Parallel hierarchical graph-scheme that permits the greatest common divisor of $N = 8$ non-negative integers to be found

Fig. 9 Interaction with the circuit that computes the greatest common divisor of eight unsigned integers

```
Greatest Common Divisor - ZyBo
Insert Number 8 numbers:
 1 -> 152
 2 -> 38
 3 -> 209
 4 -> 133
 5 -> 95
 6 -> 57
 7 -> 247
 8 -> 171
waiting...
..............
GCD = 19
```

Unsigned integers entered by the user

The computed greatest common divisor

described by PHGS in Fig. 8. Possible results of interaction from the SDK console are demonstrated in Fig. 9.

At the beginning, the operands A, B, C, D, E, F, G, and H are examined and if there is at least one zero operand then the subsequent steps are not executed and the result is assigned to 0. If all the operands are not equal to zero then 4 modules Z_1 with different arguments are activated at the same time. As soon as all of them terminate, the results of these modules are used as operands for two new invocations of Z_1 also running in parallel. The final result is produced in the single module Z_1. In [3] there are two complete synthesizable VHDL specifications that describe the hardware circuit that implements the algorithm in Fig. 8. The first specification (entity Parallel_HFSM_iterative)) corresponds to the C function discussed above. The second specification (entity Parallel_HFSM_recursive) is based on a recursive C function given in [3]. Thus, there might be recursive calls in all modules Z_1 running in parallel. The modules Parallel_HFSM_iterative and Parallel_HFSM_recursive are given in [3] (see Sect. 5.4 in [3]) and can also be downloaded from http://sweet.ua.pt/skl/Springer2014.html).

Our example uses four address ranges [15] and respectively four chip select signals with one chip enable signal for each address pair. Let us look at the following constants:

```
constant C_ARD_ADDR_RANGE_ARRAY: SLV64_ARRAY_TYPE := (
    X"0000_0000_0000_0000", -- this pair is used for 8
                 -- 32-bit operands: A, B, C, D, E, F, G, H
    X"0000_0000_0000_001F",
    X"0000_0000_0000_0020", -- this pair is used for the
         -- 32-bit result, i.e. for the greatest common
    X"0000_0000_0000_0023", -- divisor of the operands A,
          -- B, C, D, E, F, G, H
    X"0000_0000_0000_0024", -- 32-bit status (for
                     -- overflow and ready signals)
    X"0000_0000_0000_0027",
    X"0000_0000_0000_0028", -- 32-bit control (for enable
        -- and reset signals)
    X"0000_0000_0000_002B");
constant C_ARD_NUM_CE_ARRAY : INTEGER_ARRAY_TYPE := (
    0 => 1,
    1 => 1,
    2 => 1,
    3 => 1);
```

The complete project that includes hardware and software modules is available at http://sweet.ua.pt/skl/TUT2014.html. Additional details may also be found in [15]. Verification of the project demonstrates high performance. Similar experiments have been done with recursive and iterative algorithms that enable traversing binary trees from [5] to be implemented partially in software and partially in hardware.

5.2 An Example of a Parallel Hierarchical Controller

The second example explains how to execute different operations with PHFSMs/CFSMs that implement the algorithm depicted in Fig. 5. Parallel module executions are organized with the aid of the methods [3]. The main difference between HFSMs and CFSMs is in connections between the modules that are FSMs without hierarchical calls. In HFSM all links are organized through common stack memories [3] and in CFSM they are organized through semaphores [5]. The following steps have been done:

1. Incomplete in Fig. 5 PHGSs Z_4, Z_5, and Z_6 have been entirely described.
2. Nodes of the PHGSs have been marked with labels: $a_1^0, a_2^0, \ldots$ in accordance with the rules [5] (see also Fig. 5).
3. A combinational circuit for each PHFSM module $Z_0, \ldots, Z_6$ is built from memory blocks and has the structure shown in Figs. 8 and 10 of [6]. The configuration controller for memory blocks is built in a way [16].
4. The PHFSM has been synthesized and implemented in the PL with the aid of the methods [3, 6].
5. Initial configuration corresponding to the extended PHGS from Fig. 5 is done statically in the PL. Connections to the controlled devices are provided through external APSoC pins.
6. Reconfiguration that permits functionality of some modules of the PHFSM to be changed is done from software running in the PS and verified according to the methods described in Sect. 2.

We have found that reconfiguration can be done very fast. Thus, for many practical cases customization of modules may be done even during execution time. The circuit occupies less than 1 % of the PL resources, which permits many additional hardware components to be built in the same microchip.

6 Conclusion

The chapter suggests the design method for parallel logic controllers in Zynq-7000 all programmable systems-on-chip. It is proposed to model the controller by a parallel hierarchical finite state machine implemented in hardware (in the programmable logic) with additional support from software (in the processing system). The machine is composed of modules communicating with each other and managed by software, which also allows verifications and changes in the functionality of the modules applying the technique of dynamic reconfiguration. Finally, the proposed controllers provide support for modularity, hierarchy (including recursion), parallelism and run-time reconfiguration.

Acknowledgments This work was supported by National Funds through FCT—Foundation for Science and Technology, in the context of the project PEst-OE/EEI/UI0127/2014.

References

1. Sklyarov, V., & Skliarova, I. (2013). Hardware implementations of software programs based on HFSM models. *Computers and Electrical Engineering*, *39*(7), 2145–2160.
2. *Zynq-7000 All Programmable SoC Technical Reference Manual* (2014). http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf
3. Sklyarov, V., Skliarova, I., Barkalov, A., & Titarenko, L. (2014). *Synthesis and Optimization of FPGA-based Systems*. Heidelberg: Springer.
4. Zmaranda, D., Silaghi, H., Gabor, G., & Vancea, C. (2013). Issues on applying knowledge-based techniques in real-time control systems. *International Journal of Computers, Communications and Control*, *8*(1), 166–175.
5. Sklyarov, V., Skliarova, I., & Sudnitson, A. (2012). *Design of FPGA-based Circuits using Hierarchical Finite State Machines*. Tallinn: TUT Press.
6. Sklyarov, V. (2002). Reconfigurable models of finite state machines and their implementation in FPGAs. *Journal of Systems Architecture*, *47*(14–15), 1043–1064.
7. Sklyarov, V. (2002). Hardware/software modeling of FPGA-based systems. *Parallel Algorithms Application*, *17*(1), 19–39.
8. Baranov, S. (1994). *Logic Synthesis for Control Automata*. Boston: Kluwer Academic Publishers.
9. De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, Inc.
10. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*, 231–274.
11. Uchitel, S., Kramer, J., & Magee, J. (2003). Synthesis of behavorial models from scenarios. *IEEE Transactions on Software Engineering*, *29*(2), 99–115.
12. Zakrevskij, A. (1981): *Logical Synthesis of Cascade Networks*. Science, Moscow (in Russian).
13. Sklyarov, V., & Skliarova, I. (2008). Design and implementation of parallel hierarchical finite state machines. In *Proceedings of 2nd International Conference on Communications and Electronics* (pp. 33–38). Hoi An, Vietnam.
14. *LogiCORE IP AXI4-Lite IPIF v2.0. Product Guide for Vivado Design Suite* (2013). http://www.xilinx.com/support/documentation/ip_documentation/axi_lite_ipif/v2_0/pg155-axi-lite-ipif.pdf
15. Sklyarov, V., Skliarova, I., Silva, J., Rjabov, A., Sudnitson, A., & Cardoso, C. (2014). *Hardware/Software Co-design for Programmable Systems-on-Chip*. Tallinn: TUT Press.
16. Sklyarov, V., & Skliarova, I. (2007). Synthesis of reconfigurable hierarchical finite state machines. *Studies in Computational Intelligence*, Autonomous Robots and Agents (pp. 259–265). Berlin: Springer.
17. LogiCORE IP AXI Master Burst v2.0. Product Guide for Vivado Design Suite (2013). http://japan.xilinx.com/support/documentation/ip_documentation/axi_master_burst/v2_0/pg162-axi-master-burst.pdf
18. Sklyarov, V., Skliarova, I., Silva, J., & Sudnitson, A. (2014). Design space exploration in multi-level computing systems. In *Proceedings 15th International Conference on Computer Systems and Technologies* (pp. 40–47). Bulgaria.
19. *ZyBo Reference Manual* (2014). http://digilentinc.com/Data/Products/ZYBO/ZYBO_RM_B_V6.pdf
20. *ZedBoard (ZynqTM Evaluation and Development) Hardware User's Guide* (2014) Version 2.2. http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

Circuit Implementation of Parallel Logical Control Algorithms Represented in PRALU Description

P.N. Bibilo, Yu.V. Pottosin, V.I. Romanov and A.D. Zakrevskij

Abstract Software system for circuit implementation of parallel logical control algorithms LOCON-2 is described in this chapter. The system allows obtaining proper descriptions (models) in VHDL at any stage of transforming descriptions of the control algorithms. Obtaining of VHDL models provides possibility to integrate LOCON-2 with the synthesizers of logic circuits.

Keywords Digital circuits implementation · Parallel logical control algorithms · VHDL · PRALU

1 Introduction

The design of digital systems with concurrency of functioning consists of several stages. First, the initial formal model of the device behavior is tested for correctness, then it is simulated, and after that, when the model is correct, the design of appropriate logic circuits is performed in desired technological base. If the model is described in the high level design languages, such as VHDL or Verilog [1, 2], to check the compliance of high level models of digital systems with specifications for their development, the simulation is used [3]. The functional verification based on simulation demands using appropriate languages (e.g. PSL [4]), methodology [5], as well as laborious development of special testing programs [6]. So in any case, the designer must perform checking of important characteristics of the model of parallel digital system; often analyzing the state reachability graph of the system turns to be necessary [7]. Another important problem of the designing is an automatic design of logic circuits using the initial formal model of behavior of the digital system. That imposes certain restrictions on using the constructions of high level languages—only the synthesized constructions can be used in the system model [1]. PRALU language

P.N. Bibilo (✉) · Yu.V. Pottosin · V.I. Romanov · A.D. Zakrevskij
United Institute of Informatics Problems, Belarussian Academy of Sciences,
ul. Surganova 6, 200012 Minsk, Belarus
e-mail: bibilo@newman.bas-net.by

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_3

intended for description of parallel algorithms of logical control [8] is a formal language that combines successfully abilities of testing formally the initial description correctness with abilities of automated design of logic circuits. PRALU language is characterized by logical orderliness, simplicity, compactness of obtained descriptions, using binary (Boolean) input and output variables of a control device whose algorithm of functioning is specified in PRALU language. The complete description of PRALU language is in [8].

The important advantages of PRALU language in designing are developed formal methods for verification of initial PRALU descriptions of control algorithms and efficient methods for circuit implementation in the form of programmable logic arrays (PLA) with a memory organized as an RS flip-flop register. The intermediate models of such circuit implementation are parallel and sequent automata.

In this chapter, LOCON-2 system for circuit implementation of parallel logical control algorithms has been described. At any stage of transforming, there is a possibility to obtain the synthesizable VHDL descriptions. The synthesizable VHDL descriptions are those that can be used for automatic construction of logical circuits in given technological bases. Obtaining logical circuits in various technological bases can be performed by various technical synthesizers, e.g. LeonardoSpectrum [9]. LeonardoSpectrum synthesizer allows to obtain hardware implementations both in programmable logic circuits of FPGA (Field-Programmable Gate Arrays) type and custom VLSI. Converting PRALU descriptions and the intermediate descriptions (in form of parallel and sequent automata) into VHDL models is of practical interest because it allows to obtain simpler circuits and to use various technological bases.

2 Representation of Logical Control Algorithms in PRALU language

A description of a parallel algorithm of logical control in PRALU language consists of a set of sentences. Each sentence may consist of several chains. In Fig. 1, an example of PRALU description is given where every sentence, except of the fourth one, consists of one chain. The fourth sentence consists of two chains. Every chain consists of an ordered sequence of fragments. Those are initial, internal and final ones, except of the case of an elementary chain.

According to [8], a chain is called elementary if it is of the following form:

$$\mu_i : -k_i{}' \to k_i{}'' \to \nu_i \tag{1}$$

where operation $-k_i{}'$ or $\to k_i''$ may be absent. Generally, an elementary chain consists of four parts:

μ_i is the set of initial marks of the chain;
$-k_i{}'$ is the operation of waiting of event k_i';
$\to k_i''$ is the operation of action;
ν_i is the set of final marks of the chain.

```
TITLE pott1
FORMAT  PRL
AUTHOR Pottosin
DATE 30/06/14
PROJECT LOCON2_EXAMPLE1
DCL_PIN
EXT
INP
x1 x2
OUT
y1 y2
INTER

END_PIN
BLOCK pott1main                             /* comments   */
1:   -^x1*x2 > y1*^y2 -^x2 > 2.3.4;        /* sentence 1 */
2:    > ^y1 > 5.6;                          /* sentence 2 */
3.5: -x2  >  8;                             /* sentence 3 */
4:   -^x1 > ^y1 > 7;                        /* sentence 4 */
     -x1 > y2 > 9;
7:   -^x2 > 9;                              /* sentence 5 */
6.8.9:  > ^y2 -x1 > 1;                      /* sentence 6 */
END_BLOCK pott1main
END_pott1
```

Fig. 1 PRALU description pott1

We say that an elementary chain is a complete fragment. μ_i and ν_i are the sets of natural numbers. An *inner* fragment consists of a pair (operation of waiting, operation of action)

$$- k_i' \to k_i'' \tag{2}$$

An *initial* fragment

$$\mu_i : -k_i' \to k_i'' \tag{3}$$

differs from the inner one (2) by having the set μ_i of initial marks of the chain.

A *final* fragment

$$- k_i' \to k_i'' \to \nu_i \tag{4}$$

differs from the inner one (2) by having the set ν_i of final marks of the chain. If a chain is not an elementary one, it always has only one initial and one final fragment, and it may have several inner fragments. In any fragment, operation $-k_i'$ or $\to k_i''$ may be absent. All the chains in the example under consideration, with the exception of the chain that is in the first sentence, are the elementary ones.

In formulas (1)–(4), the symbols k_i' and k_i'' are elementary conjunctions of Boolean variables. The conjunctions k_i' are formed with literals of Boolean variables from the set X, and k_i'' are formed with literals of Boolean variables from the set Y. If conjunction k_i' (k_i'') is not specified in a fragment, then it is supposed to be identically equal to 1. The operation $-k_i'$ is the operation of waiting for event k_i' Executing of this operation means waiting for the event when all the variables in k_i' take values converting k_i' to 1. The operation of action $\rightarrow k_i''$ means assignment of values to variables of conjunction k_i'', such that turn k_i'' to 1. The colon is a spacer, and the arrow before ν_i denotes introducing elements into the current firing set of the chains [8].

In Fig. 1, an example of PRALU description *pott*1 [10] is given where symbol $\rightarrow$ is replaced by $>$, and symbols * and ^ are used for conjunction and complement, respectively. This is the form of PRALU description which is used in LOCON-2 system described below.

Input variables $x1$ and $x2$ form set X, and variables $y1$ and $y2$ set Y. Sentence 1 consists of one chain. There are two fragments in Sentence 1. Those are initial one

$$1 : -{}^{\wedge}x1 * x2 > y1 *{}^{\wedge} y2 \tag{5}$$

and final one

$$-{}^{\wedge}\, x2 > 2.3.4; \tag{6}$$

The initial fragment (5) has input mark 1, operation of waiting waits for variable $x1$ to take value 0 and $x2$ value 1. The operation of action is fulfilled by assigning value 1 to output variable $y1$ and value 0 to $y2$. If the variables are both in sets X and Y, they are declared in PRALU description as internal ones. There are no internal variables in the considered example. So, there are no variables in INTER section (Fig. 1).

The final fragment (6) has no input marks and operations of action. The operation of waiting means waiting for variable $x2$ to be equal to 0. The set of final marks of the fragment (6) consists of marks 2, 3, 4.

First, we will explain what the operations of waiting and action mean, and then we will explain how the algorithm is executed, i.e. how the chains are fulfilled, how they interact and what is the role of the firing set in the process.

A chain of PRALU sentence is executed if the set μ_i is a subset of the current firing set and the operation of waiting $-k_i'$ is fulfilled. Execution of a chain starts from immediate removal of μ_i from the current firing set, then the operation of action $\rightarrow k_i''$ is fulfilled and after that, the elements of ν_i are added to the current firing set immediately. At the start of the algorithm, only the mark 1 is introduced into the current firing set. The algorithm ends its work if the firing set becomes empty. During executing the algorithm, some chains may be executed at the same time (concurrently), therefore such formalism allows to describe parallel algorithms of logical control. It is considered in [8] in detail. The set of chains must satisfy the certain requirements, e.g. chains i and j with the same set of initial marks must have orthogonal conjunctions in the operations of waiting. Other requirements for correctness of initial PRALU descriptions are presented in [8]. We suppose that initial PRALU description is correct.

3 The Architecture of LOCON-2 system

The LOCON-2 program system (an abbreviation of LOgical CONtrol) is intended for designing systems of logical control based on the methodology described in [10]. LOCON-2 system consists of several subsystems (Fig. 2), each of which is oriented to working with a model of the designed control algorithm, and every model corresponds to certain stages of the design. The program system can be used jointly with the industrial synthesizer of logic circuits LeonardoSpectrum. Attention should be paid to multiplicity of the forms of description of the model of the developed control algorithm in VHDL (blocks A1–A4, RTL, OPT). In the framework of one project, all the descriptions are supposed to be functionally equivalent and differ both in terms of constructing and using peculiarities of VHDL.

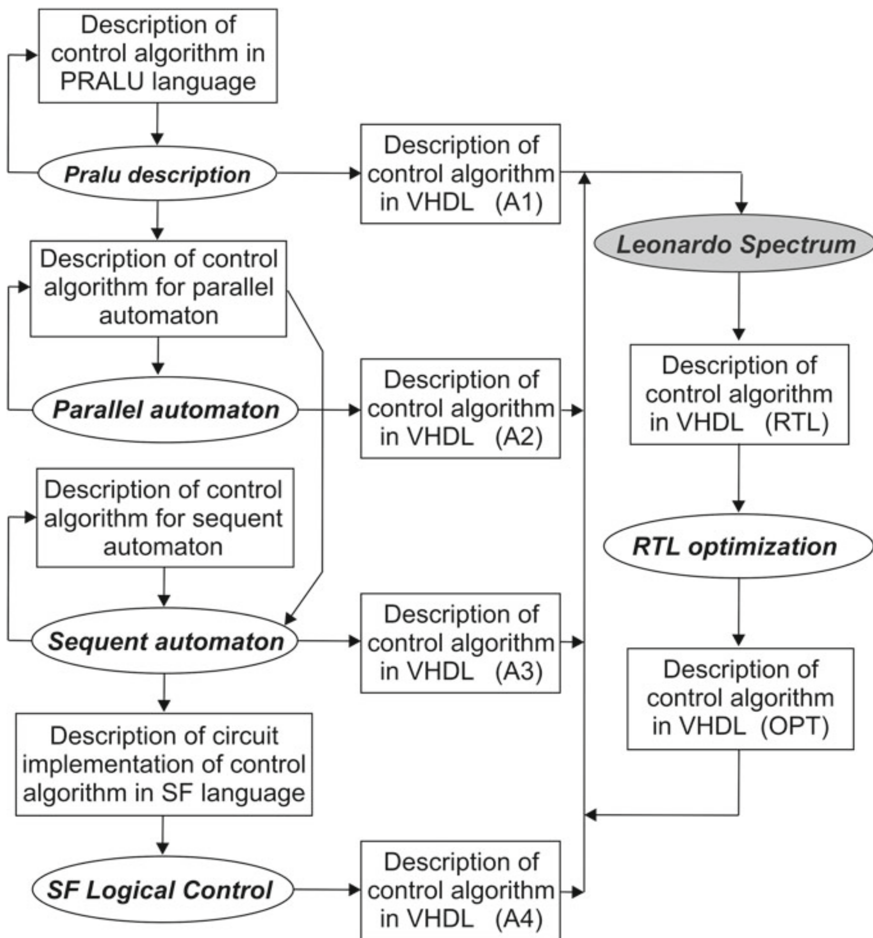


Fig. 2 Transformation of data in LOCON-2 system

4 The Technology of Design in LOCON-2

The technology of circuit implementation of control algorithms in LOCON-2 system consists of the following sequence of stages.

Stage 1. The required control algorithm is formulated in the form of description in PRALU language. Such a description is the main used model of *PRALU-description* subsystem. In the framework of this subsystem, the possibilities of convenient input and editing of algorithms in PRALU language, checking their syntax correctness and simulation are provided [10]. An algorithm being constructed and checked is converted into VHDL description [11]—the description A1 (Fig. 2) is constructed. The converting *PRALU* $\rightarrow$ *VHDL* is considered below in a more detailed way. Then the obtained control algorithm is represented as a parallel automaton.

Stage 2. The model of a control algorithm in the form of description in the language of parallel automata is the main data type for the subsystem *Parallel automaton*. The model of *parallel automaton* differs from the traditional model of finite automaton, because a parallel automaton, unlike a finite automaton, can be at several states at the same time. The transitions occur between sets of states rather than between states. In the framework of the subsystem Parallel automaton, the possibility to test important properties of a parallel automaton is provided. Those properties are self-coordination, irredundancy, recoverability, consistency and persistency.

A user can correct the initial description in terms of parallel automaton language, or return to the previous step and transform the initial PRALU description. When the model satisfying the necessary properties has been established, the corresponding representation in VHDL is constructed [12]—the description A2 (Fig. 2).

Stage 3. The subsystem *Sequent automaton* deals with the model of control algorithm as a sequent automaton [13]. The *sequent automaton* is a dynamic logical model of a discrete system with many variables that is specified formally as a set S of sequents s_i. Each sequent s_i is of the form $f_i \vdash k_i$ and determines cause-and-effect relation between an event represented by Boolean function f_i (given in disjunctive normal form, DNF) and a simple event represented by conjunctive term k_i; symbol $\vdash$ denotes this relation. The formula $f_i \vdash k_i$ is interpreted as follows: if function f_i takes value 1 at some moment, then k_i also takes value 1 immediately. By that, the values of all the variables in k_i are determined unambiguously. Working in *Sequent automaton* subsystem, a user solves the problem of state assignment of an automaton as a main problem. The success in this work provides the best circuit implementation of a considered control algorithm. It should be noted that available industrial synthesizers of logic circuits including LeonardoSpectrum are based on the *compilation (local)* principle of synthesis where each VHDL construction is replaced by its subcircuit represented at the level of RTL descriptions (*Register Transfer Level*). After that, the optimization is conducted. In state assignment of a parallel automaton that realizes several concurrent processes, another approach to hardware implementation is used that can be defined as *global* and implements jointly concurrent processes that describe functioning of the automaton as a whole.

In the conclusion of the stage, the corresponding A3 VHDL description is constructed [12]. At the same time, using the obtained sequent automaton, the model is constructed in the form of SF description that gives the functional description of PLA with the memory as an RS flip-flop register. This description can be initial for circuit implementation in various design systems that use SF description as input data. One of such design system can be, for instance, FLC system [14] that contains efficient computer programs for logic optimization. The main block of the obtained SF description is the block of combinational logic that can be optimized in the class of two-level representations of systems of functions (DNF) and in the class of BDD (Binary Decision Diagrams)—the multilevel representations constructed on the base of Shannon expansion.

Stage 4. The subsystem *SF Logical Control* allows a user to obtain VHDL description on the base of the obtained SF description. As a result, one more variant of VHDL description appears. It is A4.

Stage 5. After Stage 4 completion, the obtained descriptions A1–A4 are used as initial data for synthesizing circuits by LeonardoSpectrum synthesizer.

The process of circuit synthesis by LeonardoSpectrum synthesizer is divided into several stages, the first of which is high-level synthesis (pre-optimization). The result of this stage is an RTL description, followed by technologically independent optimization, technology mapping and increasing speed [9]. The RTL description can be obtained by LeonardoSpectrum (this distinguishes it advantageously from other synthesizers) also from the logic circuit that has been synthesized by *unmap* instruction. We call this description RTL0. Naturally, RTL0 description is equivalent functionally to the initial algorithmic VHDL description. Using RTL0 description, either ASIC or FPGA circuit can be synthesized iteratively. It is noted in design practice that a newly constructed circuit has better characteristics than a circuit that is constructed directly from initial VHDL description. So, LeonardoSpectrum synthesizer allows to decrease the circuit complexity by repeated (iterative) synthesis [14]. Analysing the obtained results of the synthesis in each of the intermediate descriptions, a user can choose the best solution and obtain the corresponding RTL description.

Stage 6. Using the RTL subsystem optimization, a user performs the optimization of RTL (description OPT at Fig. 2). The obtained description is provided again to LeonardoSpectrum synthesizer either to carry out the final synthesis or to execute the next stage of iterative synthesis [14].

5 Converting *PRALU* → *VHDL*

The way of transition from PRALU description to VHDL model described below is based on interpretation of PRALU description as a network of component finite automata that function concurrently and interact [11]. Any component automaton corresponds one-to-one to a sentence of PRALU description. The component automata interact via the common memory elements, RS flip-flops. Every RS flip-flop corresponds one-to-one to a mark from set M of all marks of the algorithm. The flip-flop

stores current (for a given cycle of discrete time) values of firing signals. Every mark i of PRALU description is put in correspondence to Boolean variable z_i. The value 1 of z_i means that mark i is in the firing set of the current cycle. The network is a synchronous circuit. The signals *clk* (*clock signal*) and *rst* (*setting into initial state*) are introduces into the circuit. These signals are supposed in the model of PRALU description. The signals *clk* and *rst* are common for all the elements of the network of component automata and memory elements. New values of the input variables of the PRALU description are supposed to be set (or not to change) in one cycle. The values of output variables do not change in one cycle. The transition to the next cycle is performed by leading edge of *clk*. The network is put into the initial state (initialized) by the value 1 of *rst*.

The network of finite automata corresponding to PRALU description *pott*1 given in Fig. 1 is shown in Fig. 3. In order not to encumber Fig. 3, the signals *clk* and *rst* are not shown in it. They are the input signals for all elements of the network.

Every sentence of PRALU description corresponds to a finite automaton. So, there are six component finite automata, *pred*1, ..., *pred*6, in the circuit (Fig. 3) for six sentences (Fig. 1). Every mark of PRALU description corresponds to a memory. element—an RS flip-flop. There are nine flip-flops because there are nine marks in the description. The output signals $y1$ and $y2$ of the circuit at Fig. 3 are supposed to be connected as "wired OR". The removal of mark z_i from the current firing set for chains corresponds to setting value 1 to Boolean variable at the input R (reset) of ith RS flip-flop. The insertion of mark zi into the current firing set corresponds to setting value 1 to Boolean variable at the input S (set) of ith RS flip-flop. Since removal and insertion of marks can be carry out in various component automata, the corresponding outputs of the automata are connected as "wired OR". In the example of Fig. 3 the outputs *n_z*9_1 of component automata *pred*4 and *pred*5 are connected as "wired OR" because final mark 9 is both in sentence 4 and sentence 5:

4: $-\hat{}x1 >\hat{}\ y1 > 7;$
$\quad -x1 > y2 > 9;$
7: $-\hat{}x2 > 9;$

Note that supplying signal 1 to both inputs R and S is inadmissible. If the signal 1 is supplied to both inputs, R and S, of RS flip-flop, it means that the circuit implementation of the parallel algorithm of logical control is incorrect. To obtain the correct description, the semantic debugging and repeated simulation of initial PRALU algorithm are needed.

To describe a finite automaton corresponding to a sentence, the set of input variables, the set of output variables, the set of internal states, the state-transition function, the output function and the initial state of the automaton must be given. We use the transition graph (state diagram) of automaton as a form of specifying a finite automaton. The transition graphs of the component automata for six sentences of PRALU description *pott*1 are shown in Fig. 4a–f. Note that the component automata are Mealy's automata.

Let us consider the forming of the component automaton *pred*1 in the example of sentence 1:

1: $-\hat{}x1 * x2* > y1 *\hat{}\ y2 -\hat{}\ x2 > 2.3.4;$

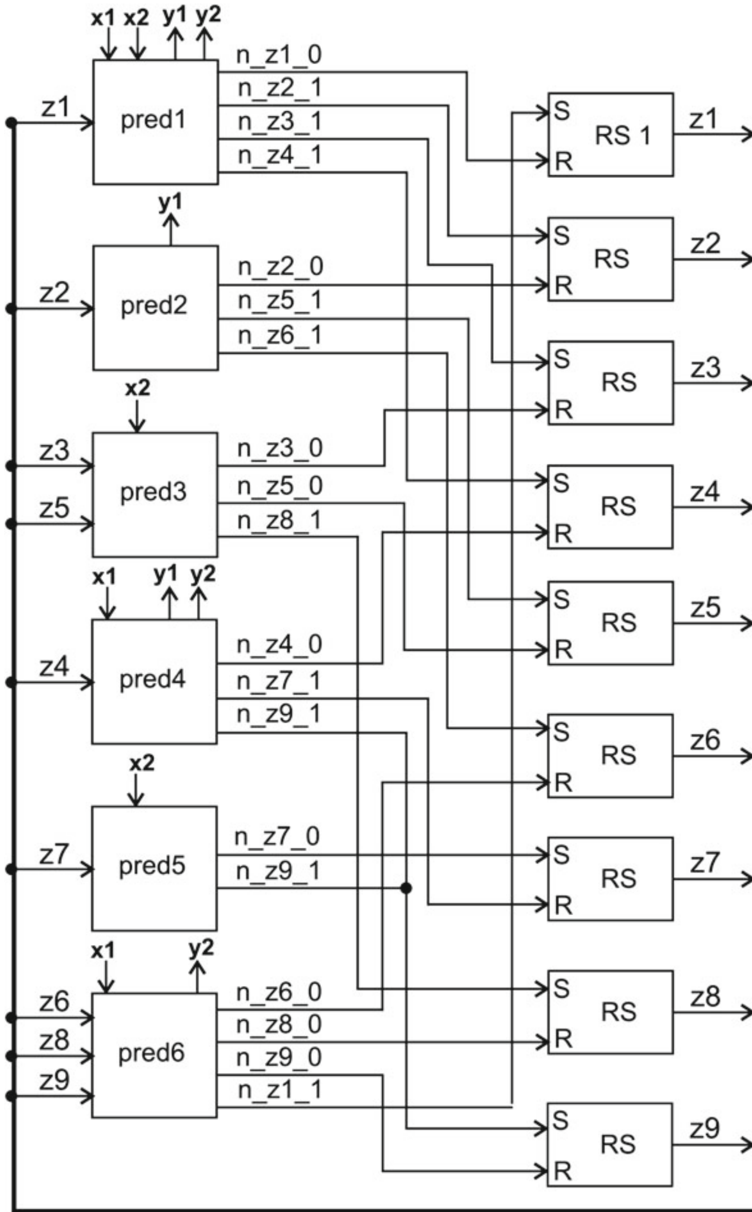


Fig. 3 Representation of PRALU description as a network of finite automata

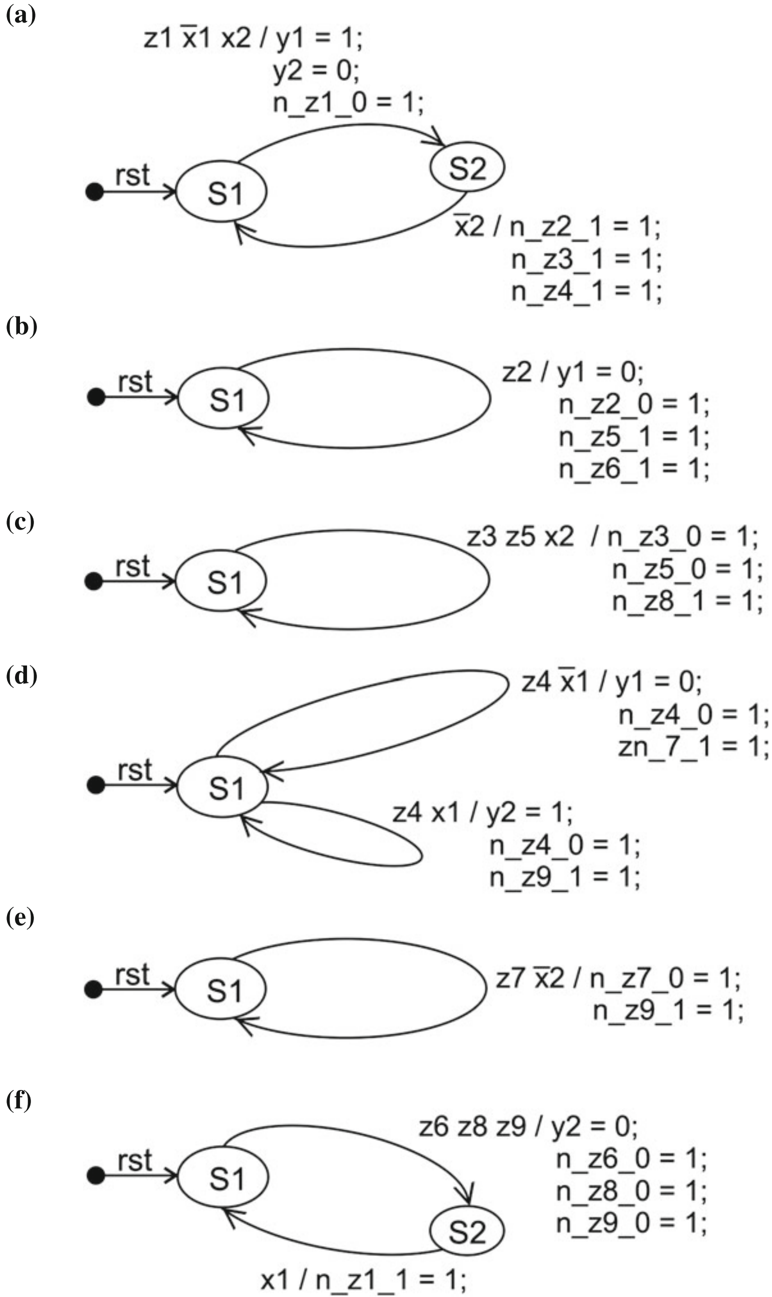


Fig. 4 Transition graphs of component automata corresponding to PRALU sentences

The set of *input* variables of an automaton is formed from the set of Boolean variables that are in operations of waiting of the considered PRALU sentence, and Boolean variables corresponding to initial marks of chains of the sentence. By default, the variables *clk* and *rst* corresponding to clock signal and setting to the initial state are added to the set of input variables of any component automaton.

The set of input variables of automaton *pred*1 consists of $x1$, $x2$, *clk*, *rst* and $z1$. The variables $x1$ and $x2$ are in the operations of waiting, and the variable $z1$ is the only mark of sentence 1. The set of output variables of an automaton is formed from the set of Boolean variables that are in operations of action, and variables corresponding to setting value 0 for initial marks of the sentence and value 1 for all the final marks of all the chains of the sentence.

In the example, the set of output variables of automaton *pred*1 consists of $y1$, $y2$, n_z1_0, n_z2_1, n_z3_1 and n_z4_1. The variables $y1$ and $y2$ are in the operations of action. The variable n_z1_0 corresponds to setting value 0 to mark $z1$. Remember that if the first elementary chain of sentence 1 is executed, then in the next cycle the mark 1 must be excluded from the firing set. That corresponds to setting the flip-flop to state 0, and this, in turn, demands supplying value 1 to input R of RS flip-flop. The variables n_z2_1 and n_z3_1 are the variables of setting value 1 for the final marks of the sentence. Figure 3 shows input and output variables of automaton *pred*1 (and other automata). The set of states of an automaton corresponding to the sentence is formed after dividing all the chains of the sentence into fragments. The beginning of a fragment corresponds conditionally to a state. Since the beginnings of all the chains coincide, the initial state $s1$ is single and common for all chains of the sentence. Then, the beginning of each fragment of a chain is put in correspondence with a state. Figure 4 is the general view of the transition graph. Each chain of a sentence corresponds to a sub-graph as a circuit. In the example, the transition graph of the component automaton corresponding to sentence 4 has two circuits, because sentence 4 of PRALU description *pott*1 consists of two chains that are elementary.

In the example under consideration, sentence 1 consists of only one chain consisting of two fragments. Consequently, the automaton *pred*1 has two states, $s1$ and $s2$ (Fig. 4a).

The transition and output functions are given on arcs connecting the vertices of the transition graph. Every vertex of the transition graph corresponds to a state of the component automaton. At every arc of the transition graph, the condition of transition and the value of the output function of the automaton are given that are disjoint with slash. Each transition between states corresponds to a fragment of a chain. The values of variables in the operation of waiting form the condition of transition. The values of the output functions are determined by the operation of action.

For an initial fragment, the values of Boolean variables corresponding to the initial marks of the fragment are introduced in the condition of transition. The output variables corresponding to initial marks of the fragment take value 0 after the transition from the state $s1$ is executed.

For final fragment, the output variables correspondent to final marks of the fragment take value 1 during the transition to the state $s1$. Let us suppose that the component automaton moves necessarily to the initial state $s1$ when the final fragment of the chain is executed.

Our example shows that the transition $s1 \rightarrow s2$ is executed (Fig. 4a) if the conjunction $z1\bar{x}1x2$ is equal to 1. This transition corresponds to the initial fragment (5). If $z1 = 1$, $x1 = 0$, $x2 = 1$ in the current cycle, then in the next cycle the output variables will have the following values: $y1 = 1$, $y2 = 0$, $n_z1_0 = 1$. Farther, the transition $s2 \rightarrow s1$ corresponds to the final fragment (6). The condition of the transition is $x2 = 0$, and the output values are $n_z2_1 = 1$, $n_z3_1 = 1$, $n_z4_1 = 1$ (That means assigning value 1 to the variables $z2$, $z3$, $z4$, corresponding to the marks 2, 3, 4).

So, according to any sentence of PRALU description, the corresponding finite automaton with abstract state can be constructed. VHDL models of parallel and sequent automata are obtained quite simply as it is described in [12].

6 Conclusion

The LOCON-2 software system allows to implement parallel logical control algorithms in PLA devices with memory as well as converting descriptions into VHDL. This provides a possibility to use the synthesis in various technological bases, including replacing PLA by a circuit of library elements or a circuit in the basis of FPGA, using the industrial synthesizer of logical circuits LeonardoSpectrum. The possibility of extracting the block of combinational logic enables to use efficient computer programs for logical optimization of systems of Boolean functions represented as DNF or BDD. The results of experiments on circuit implementation in various technological bases have shown that joint application of LOCON-2 and LeonardoSpectrum allows to obtain better results of circuit implementation of parallel logical control algorithms than a separate use of LeonardoSpectrum synthesizer.

References

1. Polyakov, A. K. (2003). *Languages VHDL and VERILOG in designing digital hardware*. Moscow: SOLON-Press (in Russian).
2. Perry, D. L. (2002). *VHDL: Programming by example* (4th ed.). New York: McCraw-Hill.
3. Hahanov, V. I., Hahanova, I. V., Litvinova, E. I., & Guz, O. A. (2010). *Design and verification of digital systems in chips Verilog & SystemVerilog*. Harkov: HNURE (in Russian).
4. Eisner, C., & Fisman, D. (2006). *A practical introduction to PSL*. Heidelberg: Springer.
5. Lohov, A. (2010). Contemporary methods for functional verification of digital HDL designs: methodology ABV, libraries OVL and QVL. *Sovremennaya elektronika*, no. 1, 56–59 (in Russian).
6. Spear, C., & Tumbush, G. (2012). *SystemVerilog for verification. A guide to learning the testbench language features*. Heidelberg: Springer.

7. Karatkevich, A. (2007). *Dynamic analysis of petrinet-based discrete systems* (Vol. 365). Lecture Notes in Control and Information Sciences Berlin: Springer.
8. Zakrevskij, A. D. (1999). *Parallel algorithms of logical control*. Minsk: ITK NAS of Belarus. M: UPCC (2nd ed.), 2003 (in Russian).
9. Bibilo, P. N. (2005). *Design systems for integral circuits based on VHDL. StateCAD, ModelSim, LeonardoSpectrum*. Moscow: SOLON-Press(in Russian).
10. Zakrevskij, A.D., Pottosin, Yu.V., Vasilkova, I.V. & Romanov, V.I. (2000). Experimental system of automated design of logical control devices (pp. 216–221) *Proceedings of the International Workshop "Discrete Optimization Methods in Scheduling and Computer-aided Design"*. Minsk: Republic of Belarus
11. Bibilo, P. N. (2006). Representation of PRALU descriptions of parallel logical control algorithms in VHDL. *Mikroelektronika*, *4*(35), 306–320 (in Russian).
12. Bibilo, P. N. (2005). Description of Parallel and Sequent Automata in VHDL. *Informatika*, no. 1, 68–75 (in Russian).
13. Zakrevskij, A., Pottosin, Yu., & Cheremisinova, L. (2008). In Keevallik, A. (Ed.), *Design of logical control devices* (p. 304). Tallinn: TUT Press.
14. Bibilo, P. N., & Romanov, V. I. (2011). *Logical design of discrete devices using the production-frame model of knowledge representation* (p. 279). Minsk: Belarus Navuka (in Russian).

Effective Partial Reconfiguration of Logic Controllers Implemented in FPGA Devices

Remigiusz Wiśniewski, Monika Wiśniewska and Marian Adamski

Abstract A method of partial reconfiguration of logic controllers implemented in FPGA is presented in the chapter. Only the control memory content is replaced while the rest of the system is not modified. The logic synthesis and implementation are performed only once. Therefore, such a realisation highly accelerates the whole prototyping process. The performed experiments showed that the original bit-stream that is sent to the FPGA can be reduced even over 500 times.

Keywords Logic controllers · Microprogrammed controllers · Partial reconfiguration · Memory · Control unit · Implementation · Field Programmable Gate Arrays (FPGA)

1 Introduction

In the traditional methods of prototyping of logic controllers [6, 7, 10, 11] the system is usually implemented with logic elements of a Field Programmable Gate Array (FPGA) [4, 12]. However, the latest FPGAs offer additionally blocks of dedicated memory that are integrated with the device [13]. Therefore, the decomposition of the controller into addressing module and memory becomes more popular. In the *microprogrammed controller*, the first part (*addressing module*) is in charge of proper addressing of microinstructions which are stored in the second module of the controller (*control memory*).

R. Wiśniewski (✉) · M. Wiśniewska · M. Adamski
Faculty of Electrical Engineering, Computer Science and Telecommunications,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: r.wisniewski@iee.uz.zgora.pl

M. Wiśniewska
e-mail: m.wisniewska@wiea.uz.zgora.pl

M. Adamski
e-mail: m.adamski@imei.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_4

In microprogrammed controllers, the control memory can be implemented with dedicated memories of the FPGA [20]. Furthermore, such a realisation allows relatively easy replacement of the content of the control memory. We show the idea of partial reconfiguration of the microprogrammed controllers, where the designer is able to modify only a few microinstructions of the controller. In the case of traditional implementation, the whole content of the FPGA ought to be replaced. The proposed idea of partial reconfiguration of the microprogrammed controllers permits to change only the content of the control memory while the rest of the system is not modified.

2 State of an Art

2.1 *Microprogrammed Controllers*

A control unit may be generally decomposed in two ways. The first one is *functional decomposition* [7, 16]. Here the decomposed based on the internal functions and states of the controller [20]. The second method is *structural decomposition* [1, 5, 20] where the task of decomposition is solved thanks to modification of the structure of the control unit. Both ideas of the decomposition lead to the *microprogrammed controllers* [19].

In the microprogrammed controller, the control unit is decomposed into two main parts. The first one is in response of microinstructions addressing. It is a simplified finite state machine [17]. The second part holds and generates the microinstructions. Such an implementation allows to minimize the number of logic elements in the destination programmable device and to apply partial reconfiguration [20].

The structure of a typical microprogrammed controller is presented in Fig. 1. Based on the input conditions (X), an addressing module (AM) forms the excitation functions T for the counter (CT), which selects (address A) the proper microinstruction (Y) from the memory (CM).

The main benefit coming from the realisation of the controller as a microprogrammed controller is the possibility of implementing the memory with dedicated

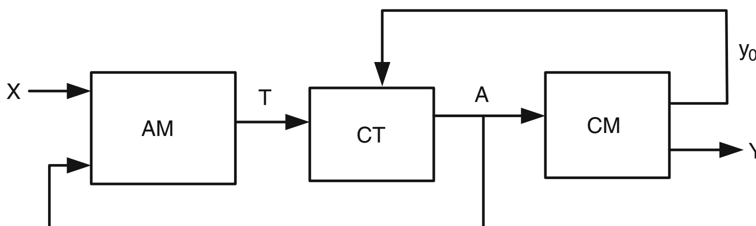


Fig. 1 Structure of a microprogrammed controller

memory blocks of an FPGA [20]. Other blocks of the prototyping system are implemented with logic blocks (flip-flops and LUT elements) of the FPGA [2, 9, 12–14]. Such an idea leads to reduction of the number of logic blocks in comparison with the realisation of the controller as a traditional finite state machine, and thus the designer can allocate a wider area of the FPGA for other blocks of the prototyping system. The effectiveness of the microprogrammed controllers is especially high if the controller interprets a linear flow-chart [5, 20]. Such a flow-chart contains 75 % of operational vertices or includes long linear chains (segments) of operational vertices.

The second advantage of the microprogrammed controller is the possibility of selecting the implementation method of control memory. The designer can decide if the module CM should be realised with logic blocks or with dedicated memory blocks [13]. It is important especially in the case of designs, which consume a large area of memory. Then the whole controller is implemented with logic blocks of the FPGA.

Finally, the microprogrammed controllers allow to apply the idea of *partial reconfiguration* [20]. In this case, only a part of the control memory can be replaced while the rest of the system remains untouched [3, 23]. The concept of partial reconfiguration is widely described in the next section.

2.2 Partial Reconfiguration of the FPGA devices

Partial reconfiguration of FPGA devices is a relatively new idea. Therefore, not all programmable devices offer the reconfiguration of part of their resources. Such a solution refers especially to devices by Xilix, Altera and Atmel [3, 23]. Although the idea of partial reconfiguration seems to be the same for various vendors, there could be technological differences. The method presented in the chapter and all further descriptions refer to partial reconfiguration proposed by Xilinx [21, 23]. In particular, the device $XC2VP30$ from the Virtex II Pro family was selected as the representative one [22]. Such a device was chosen due to its application of partial reconfiguration in practise (see Sect. 4).

From the viewpoint of the functionality of design, partial reconfiguration can be divided into two groups: *dynamic* and *static* reconfiguration. The fist one—also known as active partial reconfiguration—permits to change part of the device while the rest of the FPGA is still running. In the second solution the device is not active during the reconfiguration process. While the partial data is sent into the FPGA, the rest of the device is stopped (in the shutdown mode) and brought up after the configuration is completed [20].

The *difference-based partial reconfiguration* can be used when a small change is made in the design [21]. It is especially useful in the case of changing LUT functions or the dedicated memory blocks (like BRAMs) content. The partial bit-stream contains only information about differences between the current design structure (that resides in the FPGA) and the new content of the FPGA.

All research results and experiments presented in the chapter are based on static difference-based partial reconfiguration. This method was chosen because of the structure of the microprogrammed controllers. Difference-based partial reconfiguration allows to change the content of control memory at the implementation stage. Therefore, most steps of the prototyping flow can be omitted. Moreover, the designer can prepare more than one partial bit-streams with alternative versions of the content of the control memory. They can be very easily switched in the FPGA (the full bit-stream is sent only once).

Note that Xilinx recently had changed the reconfiguration ideas, joining them into the single designing-flow (accessible from the tool called *Plan Ahead*). All the researches and experiments described in this chapter had been done with the *ISE Xilinx 10.1* tools and are based on the *difference-based partial reconfiguration* concept presented in [21].

2.3 Mechanism of Partial Reconfiguration of Xilinx FPGAs

Figure 2 shows the basic structure of a typical FPGA device by Xilinx. The main elements of the device are *configurable logic blocks (CLBs)*, which create a matrix of connected blocks. Each CLB contains the logic elements called *slices*. Furthermore, each slice is built from *look-up tables (LUTs)*, which perform the logic functions.

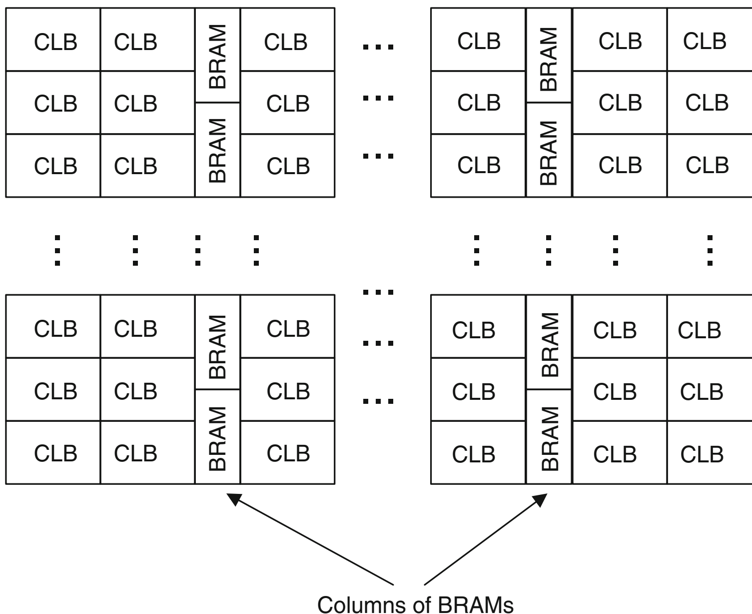


Fig. 2 Structure of the FPGA device

Depending of the device family, there could be two or more LUTs inside one logic slice (i.e. Virtex-II has two LUTs inside one slice while in Virtex-4 there are four LUTs in a single slice). All logic elements of the microprogrammed controller such as the addressing module and the counter are implemented using CLBs. Moreover, the FPGA contains dedicated memory blocks called *block-RAMs (or just BRAMs)*.

Block-RAMs are organized in *columns*. The number of columns and BRAMs in each column is different and depends on a particular FPGA. For example, the device XC2VP30 (Virtex II Pro family) contains 136 dedicated memories. They are grouped into eight columns organized as 2×20 (two columns containing 20 BRAMs), 2×18, 2×16 and 2×14 [22]. Additionally, each BRAM is divided into *lines* (called *INITs*). Lines are used for the initialization, configuration and partial reconfiguration of the block. There are 64 lines per each BRAM (counted hexadecimally from *INIT_00* to *INIT_3F*).

Both full and partial bit-streams that are used for the configuration of the device consist of *frames*. Each frame contains a portion of information about the design ought to be implemented. In the case of partial reconfiguration, only different frames are sent to the FPGA. What is very important, partial reconfiguration of Xilinx devices (especially Virtex-II family) operates on the whole column of BRAMs. This means that the modification of one microoperation (single output of the controller) in one BRAM causes the reconfiguration of all dedicated memories that belong to the same column. In the case of the XC2VP30 device, each column of BRAMs is divided into 64 frames. One frame corresponds to one line (INIT) in all BRAMs in the column (for example, the modification of two frames means the reconfiguration of two lines

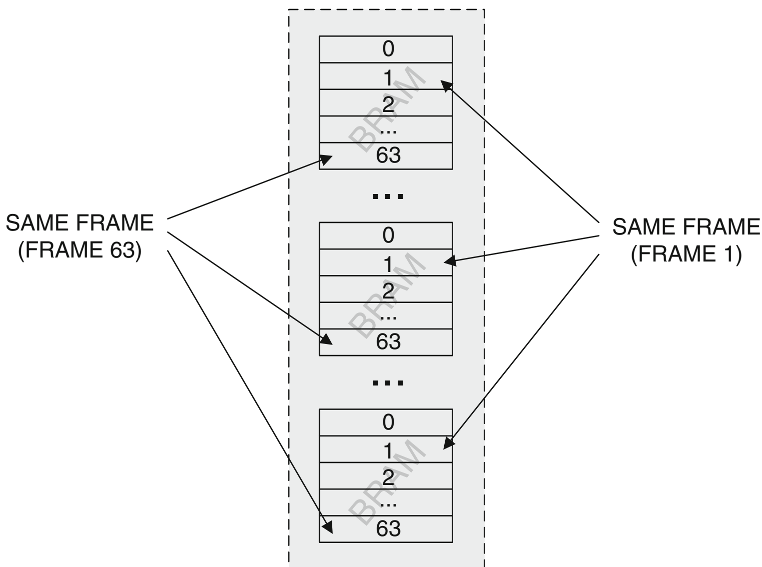


Fig. 3 Organization of BRAMs

in all blocks that belong to the column). Therefore, each frame contains a portion of information about all BRAMs that are organized in the column (Fig. 3).

The next section presents the current prototyping flow of microprogrammed controllers. Such a design process does not include the idea of partial reconfiguration. Therefore, a forthcoming section introduces a modified prototyping flow based on partial reconfiguration of microprogrammed controllers implemented in the FPGA.

3 Traditional Prototyping Flow of Control Units

In order to show the idea of partial reconfiguration of microprogrammed controllers implemented in the FPGA, the traditional prototyping flow will be presented. Figure 4 shows the design process of a typical digital system [8, 10, 15], which can be applied in the case of the microprogrammed controllers prototyping flow.

At the beginning, the specification and structure of the microprogrammed controllers ought to be prepared [5]. Next, the system may be designed according to the following steps:

1. Description of the microprogrammed controller prepared with HDL languages. At this stage, all modules (addressing module, counter and control memory) of the further control system are created. Very often hardware description languages (HDLs) like Verilog or VHDL are applied [18, 24]. The specification of the control memory content is not required at this step, although the designer can specify initial values for the controller.
2. Logical synthesis of the design. The synthesis process converts the design described with HDLs into the gate level. There are gates, logic blocks and

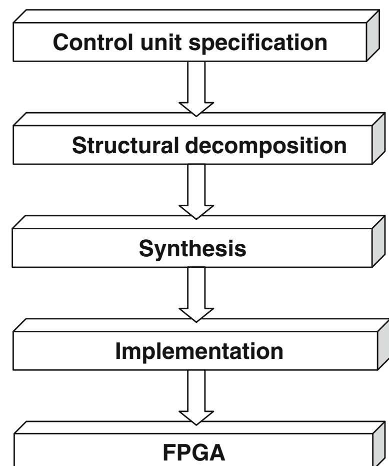


Fig. 4 Traditional prototyping flow

connections between them created as a result of synthesis (known as a "netlist"). This process is the same as in the traditional prototyping flow.

3. Logical implementation of the design. At this stage, logical implementation of the microprogrammed controller is performed. As a result of the this process, the bit-stream is produced. It contains the full description of the design that will be sent to the device to configure the FPGA.
4. Hardware implementation of the design. The FPGA is configured with the bit-stream that was produced in the previous step.

Any modification of the content of the control memory (like exchange of a single microoperation) requires repeating the full prototyping flow. Therefore, if there is a need to implement another version of the controller, all steps ought to be performed, even if the designer wants to change only one bit of the control memory.

The next section shows the idea of the prototyping flow of the microprogrammed controllers. Presented method is based on partial reconfiguration of FPGA devices.

4 Partial Reconfiguration of Microprogrammed Controllers Implemented in the FPGA

The prototyping flow for the microprogrammed controller that should be prepared for further reconfiguration is similar to the traditional prototyping process. Therefore, at the beginning, the design should be described with hardware description languages. Then it should be verified to avoid any functional errors. After the verification, the design is synthesized. The difference between the proposed and traditional prototyping flows is the implementation process. At this step, the content of further control memory is prepared. As the result of the implementation process, the configuration bit-stream is created. It contains full information about the configuration of the target FPGA. Therefore, the size of the file is respectively large. That also means long FPGA configuration time [20].

The method of partial reconfiguration of a microprogrammed controller includes the following steps (Fig. 5):

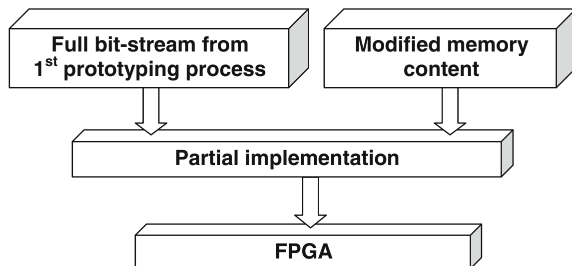


Fig. 5 Modified prototyping flow including partial reconfiguration

1. Description of the microprogrammed controller prepared with HDL languages. This step is performed in the same manner as in the traditional prototyping flow. Next, the controller should be verified in a software simulator. This allows avoiding most functional errors in the design.
2. Logical synthesis of the design. This step is the same as in the traditional prototyping flow.
3. Formation of the control memory content. Now the content of the control memory is created. The designer can prepare as many versions of the control memory content as it is necessary.
4. Logical implementation of the first version of the design. As the result of the logical implementation process, the initial bit-stream is produced. It contains the first description of the design that will be sent to the device to configure the FPGA.
5. Hardware implementation of the design. At this step, the FPGA is configured for the first time. Therefore, the whole description of the device must be specified in the bit-stream.
6. Modification of the control memory content. At this stage, the content of control memory should be replaced with alternative values that were previously prepared at step 3. The modification is performed during logical implementation. The content of the memory can be specified in many ways—by an *.ucf* file or via Xilinx tools like *FPGA Editor*, see [21, 23] for details.
7. Preparation of the difference bit-stream. Now the new bit-stream is created. It contains only the differences between the new version of the design and the previous one, which is already implemented in the FPGA. In fact, the bit-stream will contain only information about modified elements of control memory of the controller [23].
   Steps 6 and 7 should be repeated for each version of the control memory content that was prepared at stage 3.
8. Partial reconfiguration of the device. Using bit-streams produced in step 3, the device can be partially reconfigured. The functionality of the microprogrammed controller can be changed very easily and very fast, because only different frames between the modified and already implemented designs are sent to the FPGA.

5 Experimental Results

To verify the effectiveness and proper functionality of presented ideas, the partial reconfiguration process of microprogrammed controllers was verified in practise. The experiments were performed on the *XC2VP30* device. Such an FPGA contains 136 dedicated memory blocks organized in eight columns. Each column can be configured with 64 frames independently (one frame configures one line (INIT) in all BRAMs that belong to the column).

The analysis of the results of the experiments showed that the way of realising the control memory as a microprogrammed controller in the FPGA is very important. Figure 6 presents three variants of the implementation of a hypothetical

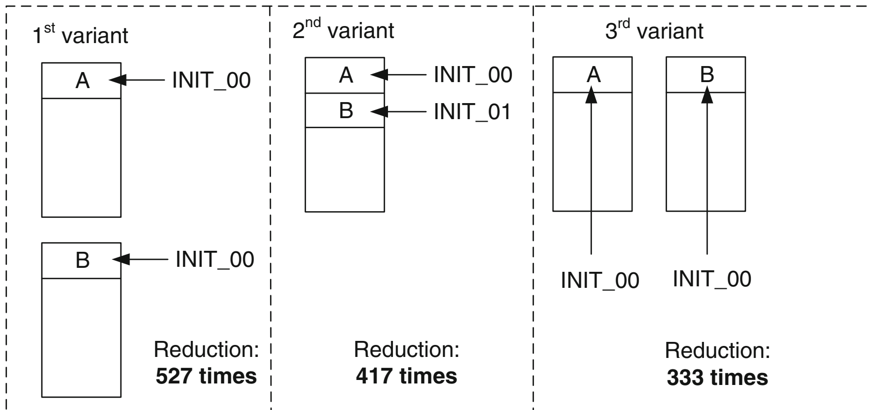


Fig. 6 Three variants of reconfiguration of two microinstructions

microprogrammed controller where two microinstructions *A* and *B* are partially reconfigured. In the first mode, both microinstructions are implemented in separate BRAMs that are placed in the same column. Both *A* and *B* are located in the line *INIT_00* of its BRAM. Therefore, during partial reconfiguration, only one frame is sent to the FPGA. Such a frame covers lines of both BRAMs, because they are situated in one column. In the second mode, both *A* and *B* are implemented in the same BRAM. However, there are two lines required, because *A* is initialized with *INIT_00* while *B* with *INIT_01*. It means that two frames are required for reconfiguration. In the third mode, *A* and *B* are implemented in two different BRAMs. Now it is not important that both microinstructions are configured with the same line (*INIT_00*), because they are located in different columns. Therefore, two frames are sent during reconfiguration.

Table 1 shows that the best results were achieved during the implementation of the first variant of the controller. Despite the fact that two lines are modified, only one frame is sent to reconfigure the device and the original bit-stream was reduced over 500 times. Very interesting results were achieved during the implementation of the two remaining variants. Both versions required two frames for partial reconfiguration.

Table 1 Results of three variants of reconfiguration of two microinstructions

Variant	Modified				Size of partial bit-stream (b)	Reduction (% of original)	Reduction (times smaller)
	BRAMs	lines	columns	frames			
1	2	2	1	1	2696	0.19	527
2	1	2	1	2	3520	0.24	417
3	2	2	2	2	4360	0.30	333

In the case of the second variant, where both microinstructions were located in the same BRAM, the bit-stream was reduced over 400 times. A worse gain was achieved in the third mode, where *A* and *B* were realised with BRAMs located in different columns.

Detailed analysis of the performed experiments indicates that the reduction of the size of the original bit-stream strongly depends on the placement of the control memory, in particular BRAM of an FPGA. The best gain is reached in the case of implementation of the control memory with BRAMs located in the same column. Partial reconfiguration of such an organization requires the least amount of configuration frames. The experiments showed that even the replacement of the content of microprogrammed controller that was implemented with 13 BRAMs (organized in one column) permits to reduce the original bit-stream by over 50 times. Furthermore, the worst results were achieved in the case of the implementation of control memory with BRAMs located in separate columns. Partial reconfiguration of the control memory that was realised with 13 BRAMs placed in eight different columns reduces the size of the bit-stream by over eight times.

Concluding, it should be pointed out that partial reconfiguration of microprogrammed controllers implemented in FPGA reduces the size of the original bit-stream even by over 500 times. In the case of controllers, where control memory ought to be decomposed into more than one BRAM, the best gain is reached during the realisation of memory with blocks located in the same column. The placement of each BRAM can be easily modified with the tools delivered from Xilinx, which additionally check routings and timing paths.

6 Conclusions

A concept of partial reconfiguration of microprogrammed controllers implemented in FPGA is presented in the chapter. Moreover, a new prototyping flow of control units is proposed. The modified design method is based on partial reconfiguration of a controller implemented in FPGA. Only the control memory content is replaced while the rest of the system is not modified. In the presented prototyping flow, logic synthesis and implementation are performed only once. Therefore, such a realisation highly accelerates the whole prototyping process. The performed experiments showed that the original bit-stream that is sent to the FPGA can be reduced even over 500 times.

References

1. Adamski, M., Wiśniewska, M., Wiśniewski, R., & Stefanowicz, Ł. (2012). Application of hypergraphs to the reduction of the memory size in the microprogrammed controllers with address converter. *Przeglad Elektrotechniczny*, *88*(8), 134–136.
2. Altera. (2008.) *Altera devices website*. California: Altera.

3. Altera. (2010). *Increasing design functionality with partial and dynamic reconfiguration in 28-nm FPGAs*. Altera.
4. Baranov, S. I. (1994). *Logic synthesis for control automata*. Boston, MA, USA: Kluwer Academic Publishers.
5. Barkalov, A., & Titarenko, L. (2009). *Logic synthesis for FSM-based control units* (Vol. 53). Lecture Notes in Electrical Engineering Berlin: Springer.
6. Bazydło, G., & Adamski, M. (2011). Specification of UML 2.4 HSM and its computer based implementation by means of Verilog. *Przeglad Elektrotechniczny*, *87*(11), 145–149.
7. Chair Brayton, R. K. (Ed.). (1993). *Sequential circuit synthesis at the gate level, Ph. D thesis*. Berkeley: University of California.
8. DeMicheli, G. (1994). *Synthesis and optimization of digital circuits*. New York: McGraw-Hill Higher Education.
9. Doligalski M. (2012). *Behavioral specification diversification for logic controllers implemented in FPGA devices: Proceedings of the Annual FPGA Conference, FPGAworld'12* (pp. 6:1–6:5), New York, USA: ACM.
10. Gajski, D. (1996). *Principles of digital design*. Upper Saddle River, NJ: Prentice Hall.
11. Grobelna, I. (2011). Formal verification of embedded logic controller specification with computer deduction in temporal logic. *Przeglad Elektrotechniczny*, *87*(12a), 47–50.
12. Łuba, T. (2005). *Synthesis of logic devices*. Warszawa: Warsaw University of Technology Press.
13. Maxfield, C. (2004). *The design warrior's guide to FPGAs*. Orlando, FL, USA: Academic Press Inc.
14. Milik, A., & Hrynkiewicz, E. (2012). Synthesis and implementation of reconfigurable PLC on FPGA platform. *International Journal of Electronics and Telecommunications*, *58*(1), 85–94.
15. Parnell, K., & Mehta, N. (2003). *Programmable logic design quick start hand book*. San Jose, CA, USA: Xilinx.
16. Rudell, R. L. (1989). *Logic synthesis for VLSI design, Ph. D thesis*. Berkeley, CA, USA: EECS Department, University of California.
17. Sentovich, E., Singh, K. J. Moon, C. W. Savoj, H. Brayton, R. K. & Sangiovanni-Vincentelli, A. L. Sequential circuit design using synthesis and optimization. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors* (pp. 328–333), Washington, DC, USA, 1992. IEEE Computer Society.
18. Thomas, D., & Moorby, P. (2002). *The Verilog hardware description language* (5th ed.). Norwell, MA: Kluwer Academic Publishers.
19. Wilkes, M. V. (1951). *The best way to design an automatic calculating machine: in Manchester University Inaugural Conference* (pp. 182–184), Manchester, UK.
20. Wiśniewski, R. (2009). *Synthesis of compositional microprogram control units for programmable devices*. Lecture Notes in Control and Computer Science, vol. 14. Zielona Góra: University of Zielona Góra Press.
21. Xilinx. (2004). *Two flows for partial reconfiguration*. Xilinx
22. Xilinx. (2007). *Virtex-II Pro and Virtex-II Pro X FGPA user guide*. Xilinx.
23. Xilinx. (2010). *Partial reconfiguration user guide*. Xilinx.
24. Zwolinski, M. (2000). *Digital system design with VHDL*. Inc, Boston, MA, USA: Addison-Wesley Longman Publishing Co.

An Application of Logic Controller for the Aerosol Temperature Stabilization

Michał Doligalski, Marek Ochowiak and Anna Gościniak

Abstract The chapter presents a logic control specification and its implementation by means of a microcontroller. The solution is dedicated to stabilisation of a spray temperature. The aim of the control is to produce a drug spray with specified temperature. The chapter draws a link between the viscosity of liquids and the droplets sizes in pneumatic inhalation process. The results indicate that the droplet size of the spray is influenced by the liquid viscosity. The liquid viscosity can be changed by temperature. Increasing the aerosol temperature decreases droplet diameters and hence increases the safety of inhaled therapy. A control system and new construction of thermostated nebulizer improving the inhalation process has been proposed.

Keywords Logic controllers · RLC · FPGA · UML · Atomization · Spray · Droplet size · Temperature stabilisation

1 Introduction

Atomization and inhalation medicines have become the major therapeutic strategy in therapy of asthma, mucoviscidosis and other diseases of the respiratory system, both for adults and children [2, 6, 9, 22]. Administration of aerosolized medicines into organism through inhalation methods is becoming more popular as a technique of therapy which has a significant advantage over the injection and oral methods [10].

M. Doligalski (✉)
Institute of Computer Engineering and Electronics, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: M.Doligalski@iie.uz.zgora.pl

M. Ochowiak · A. Gościniak
Faculty of Chemical Technology, Institute of Chemical Technology and Engineering,
Poznań University of Technology, pl. M. Skłodowskiej-Curie 2, 60-965 Poznań, Poland
e-mail: Marek.Ochowiak@put.poznan.pl

A. Gościniak
e-mail: Anna.Gosciniak@student.put.poznan.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_5

Inhalers are a kind of the devices used to spray medicine and due to their simplicity, small size, and affordable price, they are commonly used devices in medicine [3, 6, 17–21]. The mechanism of functioning of this type of devices consists in the use of compressed air energy to form drops during the drug solutions or suspensions atomization. The construction of inhaler significantly affects the atomization process. The efficiency of aerosol delivery to the respiratory tract depends on the size distribution of the aerosol particles and their physico-chemical properties, the inhalation dynamics, the type of inhalation device and correct use of the inhaler [15–17]. Viscosity of the atomized liquid is very important [3].

The viscosity of a liquid plays a key role in the atomization process and affects the droplet size and/or the droplet size distribution obtained by atomization of the liquid. Droplet size is measured by D_{32} (SMD, Sauter) or $D_{0.5}$ (MMD) [16]. D_{32} is defined as:

$$D_{32} = \frac{\Sigma N_i d_i^3}{\Sigma N_i d_i^2} \tag{1}$$

where i is the considered size range, N_i is the number of droplets in size range i, and d_i is the diameter corresponding to the centre of the range $\langle d_i - \frac{\Delta d}{2}, d_i + \frac{\Delta d}{2} \rangle$. In addition, the values of the mass median diameter (MMD, D0.5) have been determined. D0.5 is a droplet diameter such that 50 % of total liquid volume is in droplets of smaller diameter [14].

Analysing the influence of viscosity on droplet size, Dorman [8] reported that $D_{32} = f(\eta^{0.1})$. Hasson and Mizrahi [12] found that D_{32} is proportional to $\eta^{1/6}$ for values of viscosity from 1×10^{-3} to 21×10^{-3} [Pa.s].

The liquid viscosity can be reduced by increasing the temperature of the solution in the nebulizer. Pneumatic inhalators produce an aerosol with a temperature close to the ambient temperature. The aerosol is unfavorable for newborns, infants, hypersensitive patients, allergy sufferers and others with bronchial hyperreactivity. The aerosol at an elevated temperature significantly increases the safety of inhaled therapy. The new construction of thermostated nebulizer improves the inhalation process for the patients in homes.

The nebulisation process requires temperature stabilisation. Too high temperature may be harmful for the patient, unstable temperature will cause different droplets sizes production. Temperature control system should detect temperature changes and react if the temperature is below the desired value. This requirement is important both in the case of personal use and of drop size test. Personal use of the nebulizer requires cheap and compact system that could be used as an extension of the pneumatic nebulizer head.

The droplet size test requires additional temperature reports. Droplet size depends on temperature, and that is the reason why the spray temperature should be documented during test session. Ambient temperature fluctuations and other random disturbances can cause undesirable changes in temperature. Automatic temperature registration will improve and document droplet size tests.

Pneumatic nebulizers available on the market produce aerosol, temperature of which differs from ambient temperature. There are no devices that would increase

the safety of the inhalation process, producing spray at a temperature in the range from 28 to 37 °C. Mainly ultrasonic nebulizers feature the function. Pneumatic devices in order to generate thermoaerozol must be equipped with additional thermostats or the aforementioned thermal snap. The chapter presents the application of microcontroller-based logic controller to stabilize the temperature of a drug spray.

2 Logic Controller

The nebulizer temperature control system will have to control spray temperature and store control data. The control system will be implemented as a logic controller with two-step control where output signal will have only two values. The role of the unit is a fixed value control to maintain constant temperature. In the case of such control system there is no possibility to determine the equilibrium where stabilised temperature constantly has the reference value. The temperature will oscillate around the reference value. Such control system could be implemented as a reconfigurable logic controller (RLC) by means of FPGA devices or as a reprogrammable system by means of a microcontroller [23].

Implementation of the logic controllers by means of FPGA devices is very popular. Formal modelling and verification techniques [13] increase the system dependability. FPGA devices are suitable for complex control systems implementation where one controller is equipped with a number of inputs and outputs, then logic resources guarantee that even a large system can be implemented. The implementation of Ethernet interface and data logging is also possible, but it requires additional software implemented in FPGA or a hard processor. It increases system complexity, so such devices are not suitable for small projects. The configuration of the FPGA devices can be modified including reconfiguration of the active (working) device [4, 7, 24]. It increases design functionality and can reduce required FPGA resources.

The control system can be also implemented by means of a microcontroller. Such device provides serial inputs and outputs (UART, I^2C, SPI) but the number of binary inputs/outputs is limited. The implementation of the Ethernet interface or Flash-based data logging is simpler than in FPGA. Control algorithm can be implemented using Java, Python or C language. Reconfiguration (reprogramming) of the controller is also possible, it can be performed by means of using *Strategy* design pattern. For spray temperature stabilisation such implementation is not a critical requirement. The controller should have the possibility of the reference temperature configuration but it should be unchanged during measurement cycle.

The chapter presents rapid prototyped approach to logic controller implementation. The design complexity should be low and should take into account the further commercial product. The implementation time and costs are the key criteria.

The presented logic controller is implemented by means of a microcontroller. The prototype is based on a single-board computer, the Raspberry Pi credit card-sized board was used. The main advantages of the device are as follows: low price, Ethernet interface, COTS expansion modules. Also availability of free operating systems is

very important. Specified requirements are met by the selected platform, also further development is not limited. The controller current TRL level is estimated as 5, the breadboard was validated in a relevant environment.

The system will be built with temperature sensors and a heating coil disposed in the head nebulizer. The control algorithm was implemented by means of Java language. The control software is dedicated to Raspberry Pi platform. It can be also used in another system, but in that case, input/output modules should be redefined.

Nebulizers are popular both in home and hospital therapy. The controller should guarantee safety for a user, who does not always have appropriate qualifications. Such therapy is popular and effective in respect of treatment of children but, unfortunately, head tilt in children is a common problem. Also some drugs cannot be overheated. Excessive heating of a product contained in the vessel can result in inactivation and degradation of the drug, resulting in the loss of its therapeutic properties. The head can be overheated when the liquid was completely sprayed or accidentally spilled out of the head.

The presented solution implements safety procedures. The logic controller prevents head and liquid overcharging, liquid temperature limit can be also specified. When liquid temperature is higher than the specified value the heater power supply is cut-off. The controller also has to be resistant to noise, which may be generated as a result of the opening of the vessel lid, or a sudden change in the ambient temperature and supply voltage variations heater.

For the logic controller specification, Petri nets and UML state machine can be used as well [5]. UML-based formal specifications, such as machine and activity diagrams can be verified by means of model checking [11]. Figure 1 presents a logic controller state machine-based specification. The two-step control algorithm controls the temperature of the spray. Temperature measured by sensor (T1) is compared with reference value (TRef), if it is below the limit, the *Active* state is activated and output signal *Y1* is generated. A simple exception handling method is also implemented. Additional sensor (T2) measures temperature of the liquid (drug) in the nebulization

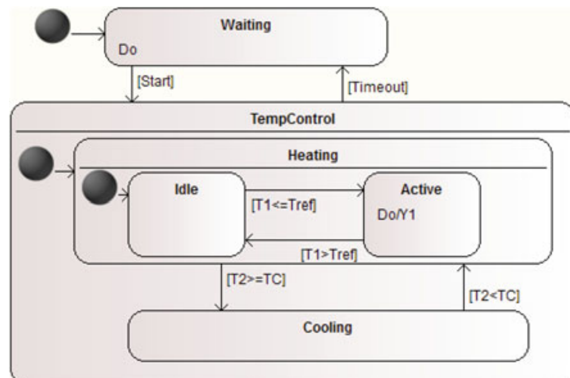


Fig. 1 State machine for the initial strategy

head. If the temperature rises above specified limit (TC), the power supply to the heater is shut off.

The third temperature sensor (T3) is not an input of the logic controller, it is used for ambient temperature monitoring. It is important at the analysis stage because ambient temperature may affect the test result.

Additionally, timers can be used to control experiments or therapy time. External rely (not used in the project) can be used to control compressor supply. The spray can be generated after drug in nebulizer head obtains required internal temperature (pre-heating). This is a next example of the controller reprogramming strategy.

3 Experimental Set-Up

For the analysis of sprays the method of collecting droplets in a cell containing a suitable immersion liquid (oil) by digital microphotography using microscope Nikon Eclipse 50i was used. The setup also includes MultiScanBase (Computer Scanning Systems II) software for visualization and image processing and Image-Pro Plus 6.0 (Media Cybernetics Inc.) for digital images processing and image analysis.

The scheme of the experimental set-up used in the study is shown in Fig. 2. The MedelJet Family inhaler with Philips Respironics Jet Pro nebulizer was used. Medel Jet Family inhaler is a pneumatic inhaler with a compressor that is applied for aerosol therapy. According to the instructions provided by the manufacturer, the rate of inhalation for this model is 0.28 ml/min and maximum pressure is 230 kPa. The test model liquid (distilled water) at different temperatures was atomized at the same ambient conditions.

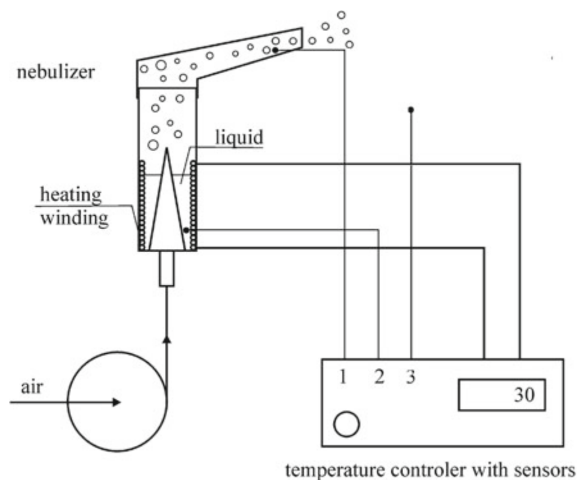


Fig. 2 Nebulization process control system

The temperature controller implemented by means of microcontroller was equipped with three waterproof probes. The following probes are used to measure the temperature of: 1—spray, 2—liquid, 3—ambient. Each probe is based on DS18B20 Programmable Resolution 1-Wire Digital Thermometer. The accuracy is ±0.5 °C in range from −10 to 85 °C.

The output of the controller is connected to the relay module. Power relay closes the heating circuit constructed of a heating coil of PFTE (Teflon) coated resistance wire. Teflon has high chemical resistance, which means that it does not react or dissolve in most compounds. Moreover, its high melting point of 327 °C allows use of Teflon in the systems at high temperature. Small surface free energy protects the material from being buried in dirt, which is one more advantage, especially important in the case of drug delivery.

Figure 3 presents control data logged by the system. Values from all three temperature inputs are presented. It shows the abnormal temperature of the liquid in the head, that was caused by improper head tipping. Exceptions handling mechanism allows to prevent over-temperature above 60 °C. The temperature can be configured remotely. Results of the measurements and control systems are stored in flash memory.

The test solution was sprayed in order to determine the diameter of the solution drops, and the droplets were captured in the liquid immersion (20–90 oil, delivered by The Oil and Gas Institute from Cracow). The microscopic images of atomized liquid were photographed by means of an Opta-Tech camera. The aerosol obtained by spraying was captured on the layer of immersion liquid which was evenly distributed on a glass plate. The spray characteristics were obtained by averaging the values of minimum 1400 drops for each of experimental fluids. The accuracy of drop diameter measurements was ±0.3 μm. The accuracy of the average diameter based on the number of analysed droplets was ±10 % [1].

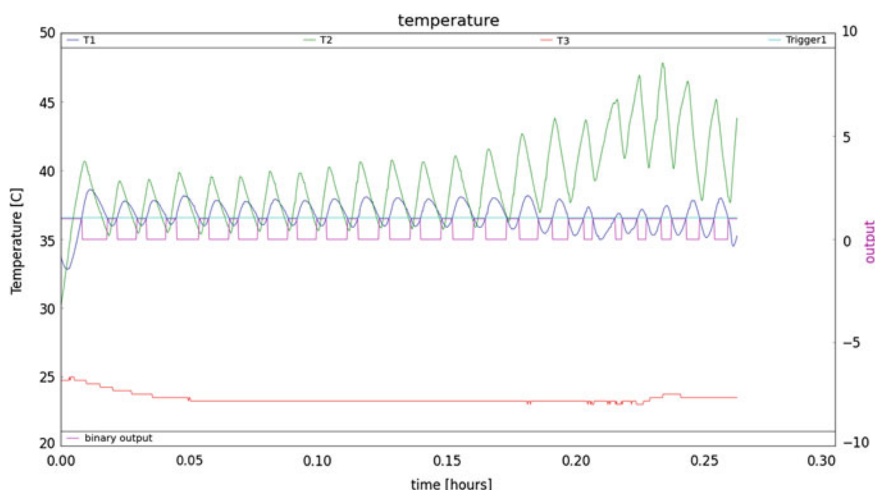


Fig. 3 Logged control data

4 Tests Results

The exemplary images of drops obtained by atomization of water are presented in Fig. 4. Comparison of the images has shown that the number of big droplets is decreasing with growth temperature of the atomized liquid.

The exemplary droplet size histograms are presented in Figs. 5 and 6. Droplets of water at 20 °C have diameters up to 18 μm and at 50 °C up to about 13 μm. The most of observed droplets have a diameter from 2 to 10 μm at 20 °C and from 1 to 4 μm at 50 °C. At higher value of temperature the histogram is tall and narrow.

Viscosities of water are respectively 0.001 [Pa.s] at 20 °C and 0.0055 [Pa.s] at 50 °C. The values of D_{32} calculated from the experimental data are respectively at 20 °C $D_{32} = 8.1$ μm and at 50 °C $D_{32} = 4.05$ μm. The results indicate that an increase of the viscosity of the solution leads to a larger droplet size. It has been shown that the D_{32} values increase with increasing of viscosity of liquids.

It means that liquids at an elevated temperature (with small viscosity) are atomized in a similar way but the atomization is easier in comparison with liquids at ambient temperature (high values of viscosity).

Aerosol spray temperature is approx. 5–10 °C lower than the temperature of the liquid in nebulizer head. The controller allows to control the temperature of the heating coil in the nebulizer head, so that the desired temperature of the liquid can be reached as soon as possible. The accuracy of temperature stabilisation is also high, it does not deviate more than ±1.0 °C. The constant-value system maintains a setpoint, responds well and quickly to compensate disturbances and their impact. The tests confirm the controller usability and its high accuracy.

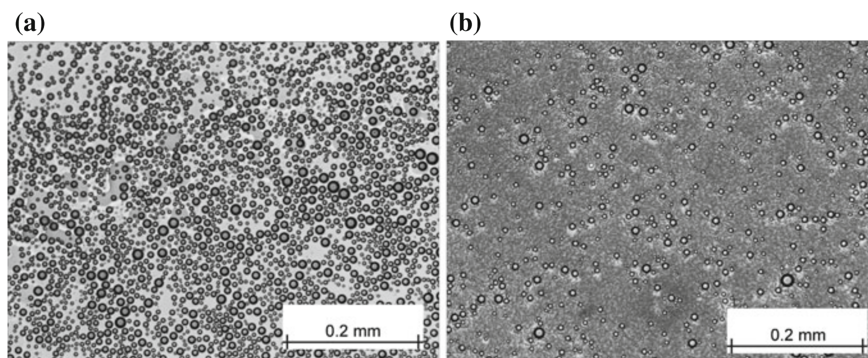


Fig. 4 Photos of the aerosol: **a** water at 20 °C; **b** water at 50 °C

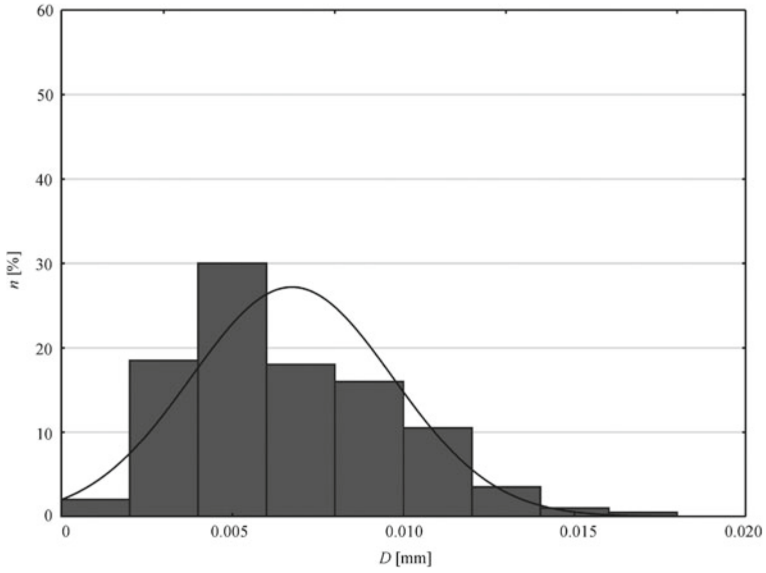


Fig. 5 Droplet size histograms based on number of the droplets, water 20 °C

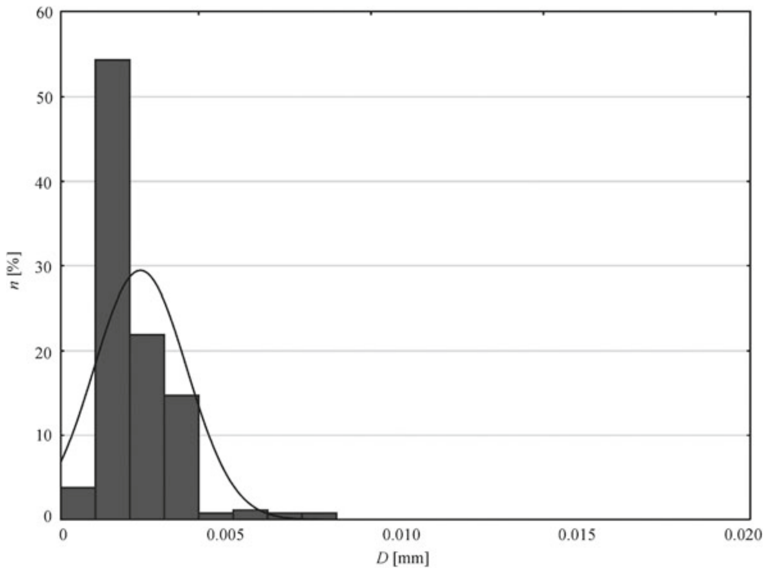


Fig. 6 Droplet size histograms based on number of the droplets, water at 50 °C

5 Conclusions

The control system for the nebulisation head temperature was presented in this chapter. The chapter also draws a link between the viscosity of liquids and the droplets size in a pneumatic inhalation process. The results indicate that the droplet size of the spray is influenced by the liquid viscosity. The liquid viscosity can be changed by temperature. The aerosol at an elevated temperature significantly decreases droplet diameters and increases the safety of inhaled therapy. Using the results of the analysis, a new construction of a thermostatted nebulizer improving the inhalation process has been proposed.

The resulting two-position control system allows to set and change the setpoint, monitoring, creating a database of measurements, as well as the presentation of the results. The temperature can be stabilised at a required level, reconfiguration of the controller also can be performed. A critical exception handling mechanism can be implemented. Presented research should be continued in order to determine the effect of temperature and viscosity of the liquid droplet diameter value for other structures of the nebulizers. The validity of microcontrollers has been confirmed but the prototype should be redesigned, especially board size should be reduced. Also the heater should be adapted to meet the requirements of hygiene.

References

1. Azzopardi, B. J. (1979). Measurements of drop sizes. *International Journal of Heat and Mass Transfer*, *22*, 1245–1279.
2. Bisgaard, H., O'Callaghan, C., & Smaldone, G. C. (Eds.). (2002). *Drug delivery to the lung*. New York: Marcel Dekker Inc.
3. Broniarz-Press, L., Ochowiak, M., Markuszewska, M., & Włodarczak, S. (2013). The effect of viscosity on the atomization process in medical inhaler (in Polish). *Chemical Engineering Equipment*, *4*(52), 291–292.
4. Bukowiec, A., & Doligalski, M. (2013). Petri net dynamic partial reconfiguration in FPGA. In A. Quesada-Arenciba, R. Moreno-Diaz, & F. Pichler (Eds.), *Computer aided systems theory—Eurocast 2013* (Vol. 8111, pp. 436–443)., Lecture Notes in Computer Science, Berlin: Springer.
5. Bukowiec, A., & Tkacz, J. (2014). Dual simulation of application specific logic controllers based on petri nets. In *Multimedia and ubiquitous engineering (MUE) : 8th international conference* (pp. 399–404)., Lecture Notes in Electrical Engineering Zhangjiajie, Chiny. (ISBN: 978-3-642-54900-7)
6. Devadason, S. G. (2006). Advances in aerosol therapy for children with asthma. *Journal of Aerosol Medicine*, *19*, 61–66.
7. Doligalski, M., & Bukowiec, A. (2013). Partial reconfiguration in the field of logic controllers design. *International Journal of Electronics and Telecommunications*, *59*(4), 351–356.
8. Dorman, R. G. (1952). The atomization of liquids in a flat spray. *British Journal of Applied Physics*, *3*, 189–192.
9. Gradon, L., & Marijnissen, J. C. M. (2003). *Optimization of aerosol drug delivery*. Dordrecht: Kluwer Academic Publisher.
10. Gradon, L., & Podgorski, A. (2004). Production of nanostructured particles for medical applications (in polish). *Inżynier Chemical Processing*, *25*, 1915–1923.

11. Grobelna, I., Grobelny, M., & Adamski, M. (2014). Model checking of UML activity diagrams in logic controllers design. In W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, & J. Kacprzyk (Eds.), *Proceedings of the ninth international conference on dependability and complex systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland* (Vol. 286, pp. 233–242)., Advances in Intelligent Systems and Computing Heidelberg: Springer International Publishing.
12. Hasson, D., & Mizrahi, J. (1961). The drop size of fan spray nozzles. *Transactions of the Institution of Chemical Engineers*, *39*, 415–422.
13. Karatkevich, A. (2007). *Dynamic analysis of Petri net-based discrete systems*. Lecture Notes in Control and Information Sciences Berlin: Springer.
14. Lefebvre, A. H. (1989). *Atomization and sprays*. New York: Hemisphere Publishing Corporation.
15. McCallion, O. N. M., & Patel, M. J. (1996). Viscosity effects on nebulisation of aqueous solutions. *International Journal of Pharmaceutics*, *130*, 245–249.
16. McCallion, O. N. M., Taylor, K. M. G., Bridges, P. A., Thomas, M., & Taylor, A. J. (1995). Jet nebulisers for pulmonary drug delivery. *International Journal of Pharmaceutics*, *130*, 1–11.
17. Moskal, A., & Sosnowski, T. R. (2009). Dynamics of aerosol pulse in a simplified mouth-throat geometry and its significance for inhalation drug delivery. *Chemical Engineering and Processing*, *30*, 545–558.
18. Nagel, M. W., Wiersema, K. J., Bates, L. S., & Mitchell, J. P. (2002). Performance of large- and small-volume valved holding chambers with a new combination long-term bronchodilator/anti-inflammatory formulation delivered by pressurized metered dose inhaler. *Aerosol Medications*, *15*, 427–433.
19. Petersen, F.J. (2004). *A new approach for pharmaceutical sprays. Effervescent atomization. Atomizer design and spray characterization. Ph. D thesis*. Department of Pharmaceutics: The Danish University of Pharmaceutical Sciences.
20. Petersen, F. J., Worts, O., Schaefer, T., & Sojka, P. E. (2004). Design and atomization properties for an inside-out type effervescent atomizer. *Drug Development and Industrial Pharmacy*, *30*(3), 319–326.
21. Pilcer, G., & Amighi, K. (2010). Formulation strategy and use of excipients in pulmonary drug delivery. *International Journal of Pharmaceutics*, *392*, 1–19.
22. Sheth, P., Stein, S. W., & Myrdal, P. B. (2013). The influence of initial atomized droplet size on residual particle size from pressurized metered dose inhalers. *International Journal of Pharmaceutics*, *455*, 57–65.
23. Tkacz, J., & Adamski, M. (2012). Logic design of structured configurable controllers. In *IEEE 3rd international conference on networked embedded systems for every application—NESEA. (2012)* (p. 6). Wielka Brytania, Liverpool.
24. Wiśniewski, R., Barkalov, A., & Titarenko, L. (2008). Partial reconfiguration of compositional microprogram control units implemented on an FPGA. In *Proceedings of IEEE east-west design & test symposium—EWDTS 08* (pp. 80–83). Lviv, Ukraine: Kharkov National University of Radioelectronics, Lviv, The Institute of Electrical and Electronics Engineers, Inc.

Symbolic Coloring of Petri Nets

Jacek Tkacz

Abstract In this chapter two methods of automatic coloring of the Petri nets supported by formal reasoning using monotone Gentzen calculus are presented. Coloring is used to determine the State Machine subnets. The colors help to validate intuitively and formally consistency of all sequential processes in the considered discrete state model. The first of presented methods is based on determination of exact transversals of the concurrency hypergraph. The second method is based on an examination of the relationship between the sets of minimum siphons and traps. Use of the Gentzen calculus allows to obtain additional information (the proof trees) from the reasoning process. The proof trees represent a formal proof of correctness of the calculation performed during the determination of the State Machine subnets. The methods presented in the chapter can be used in design of the reconfigurable logic controllers.

Keywords Petri net · Logic controllers · SM coloring · Decomposition · Formal reasoning

1 Introduction

Obtaining covering of a Petri net by the State Machine subnets is important both from the theoretical point of view [10] and in the applications such as design of parallel logical controllers [14]. We use here the terminology from [13], according to which such covering is understood as a kind of coloring of a Petri net, where a color is associated to an SM-subnet of the net.

There are many different algorithms of coloring of the Petri nets [7, 11], including:

- manual coloring during specification,
- coloring based on topological structure of the net,

J. Tkacz (✉)
Institute of Computer Engineering and Electronics,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: j.tkacz@iie.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_6

- coloring from the concurrency hypergraph performed with the use of deduction methods,
- coloring with the use of siphons and traps,
- coloring of discrete space of the net.

Most of them (except the manual coloring) can be fully automated and included into logic controller design process. In this chapter the methods based on concurrency hypergraphs and using siphons and traps will be presented.

Assigning of the colors (or symbols corresponding to them) to the places and transitions of a Petri net helps to validate consistency of the sequential processes in the Petri net under consideration. Each color corresponds to one state machine module. The rules for Petri net coloring are as follows [4, 9]:

- each place and transition must have at least one color,
- if a place has a color each of its input and output transitions must have the same color,
- input places of each transition must hold different colors,
- output places of each transition must hold different colors,
- input and output places of a transition must share the same set of colors,
- initially marked places cannot share the same colors,
- number of different colors which are shared by the places initially marked is equal to the total number of colors.

A Petri net is safe if each place of it can contain not more than one token in every reachable marking. A Petri net is live when all transitions are live. A transition t is live if for any marking M', reachable from the initial marking M_0, a sequence of transitions exists friable from M' which contains transition t [10]. A Petri net with the assigned colors, if it is additionally live and safe, is called a colored interpreted Petri net [13].

The proposed coloring methods of the Petri nets use Gentzen formal symbolic deduction. Gentzen symbolic reasoning [5] establishes a link between Boolean expressions, commonly used in a digital system design on RTL level, and the Petri net model of a Hierarchical Concurrent State Machine [1].

2 Gentzen Deduction System

The sequent is a formalized statement used for deduction and calculi [5]. In the sequent calculus, sequents are used for specification of judgement that are characteristic for deduction system. The sequent is defined as an ordered pair (Γ, Δ), where Γ and Δ are the finite sets of formulas, and $\Gamma = \{A_1, A_2, \ldots, A_m\}$, $\Delta = \{B_1, B_2, \ldots, B_n\}$. Instead of (Γ, Δ) it is a used notation with the use of turnstile symbol $\Gamma \vdash \Delta$. Γ is called the antecedent and Δ is the succedent of the sequent. The sequent $\Gamma \vdash \Delta$ is satisfiable for the valuation v iff for the same valuation v the formula $\bigwedge_{i=1}^{m} A_i \rightarrow \bigvee_{j=1}^{n} B_j$ is satisfied. In the proposed implementation of

Gentzen system there are defined ten rules of elimination of logic operators. For each operator (negation, disjunction, conjunction, implication and equivalency), there are defined two rules of its elimination. First rule is used when the operator is located in antecedent and the second one when it is located in a succedent. As an example the rule of the disjunction operator elimination will be presented. If the main logical operator in a sequent is a disjunction located in the succedent then two sequents will be produced. First sequent will contain the first comma separated argument of the disjunction in the succedent, and second sequent will contain the second argument of the disjunction also in the succedent. The elimination process is repeated, while only normalized sequents are obtained.

A normalized sequent is a sequent without any logical operators. A sequent is a tautology if it has the same formula in its antecedent and succedent. The tautology sequents could be removed from further normalization. If and only if all normalized sequents are the tautologies then the analysed root sequent is also a tautology. When one of the leafs in deduction tree is not a tautology it means that it is a counterexample for the analysed sequent. The consensus method based on logic resolution is also included into formal deduction (Gentzen cut rule).

The current version of implementation of Gentzen system "GENTZEN v6.7.2" accepts many types of logical operators, taken from Palasm (`*`, `+`, `/`, `->`, `<->`, `<+>`), VHDL (`and`, `or`, `nor`, `xor`, `and`, `not`, `<=`), Verilog (`&`, `|`, `!`, `^`), linear logic ($\oplus$, $\otimes$, $\multimap$) and NuSMV (`&`, `|`, `xor`, `xnor`, `!`). The implemented system is optimized by using the elements of the Thelen's algorithm [8]. This combination significantly reduces the proof trees and improves overall system performance. In order to obtain optimal solutions to the algorithm was also added a sequent version of the resolution algorithm [3]. Results obtained from the system can be automatically transformed to VHDL, Verilog, NuSMV (a model for model checking system), Espresso, and classic CNF and DNF forms.

3 Coloring from Concurrency Hypergraph

It is possible to deduce from a colored Petri net (Fig. 1) its equivalent representation of the controller as a transition system. It can be obtained from the reachability graph (Fig. 2) of the Petri net. The vertices of a reachability graph describe the global states of a transition system. If a Petri net is properly colored then the vertices contain all colors. There are no vertices which contain two different places with the same color. The number of colors should be minimal and equal to the maximum number of places which can be concurrently marked. In this way the transition system could be treated as an interpreted form of reachability graph (Fig. 2). The transition system can be in ten global states $M_0 - M_9$. There are superposition of maximal subsets of concurrent local states:

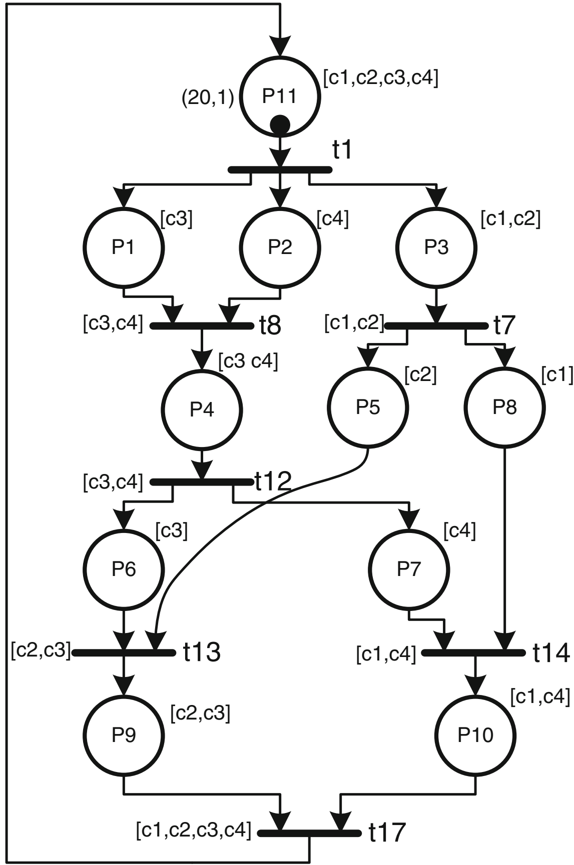


Fig. 1 Example of the Petri net

$$M_0 = \{P_{11}\};\ P_1 = \{P_1, P_2, P_3\};$$
$$M_2 = \{P_3, P_4\};\ P_3 = \{P_1, P_2, P_5, P_8\};$$
$$M_4 = \{P_3, P_6, P_7\};\ P_5 = \{P_4, P_5, P_8\};$$
$$M_6 = \{P_5, P_6, P_7, P_8\};\ P_7 = \{P_5, P_6, P_{10}\};$$
$$M_8 = \{P_7, P_8, P_9\};\ P_9 = \{P_9, P_{10}\}$$

The monotone characteristic sequent of the Petri net discrete space is as follows:

$$\begin{aligned} M \vdash\ & (P_{11}), (P_1 \text{ and } P_2 \text{ and } P_3), (P_3 \text{ and } P_4), \\ & (P_1 \text{ and } P_2 \text{ and } P_5 \text{ and } P_8), \\ & (P_1 \text{ and } P_2 \text{ and } P_5 \text{ and } P_8), \\ & (P_3 \text{ and } P_6 \text{ and } P_7), \\ & (P_4 \text{ and } P_5 \text{ and } P_8), \\ & (P_5 \text{ and } P_6 \text{ and } P_7 \text{ and } P_8), \\ & (P_5 \text{ and } P_6 \text{ and } P_{10}), \\ & (P_7 \text{ and } P_8 \text{ and } P_9), (P_9 \text{ and } P_{10}); \end{aligned}$$

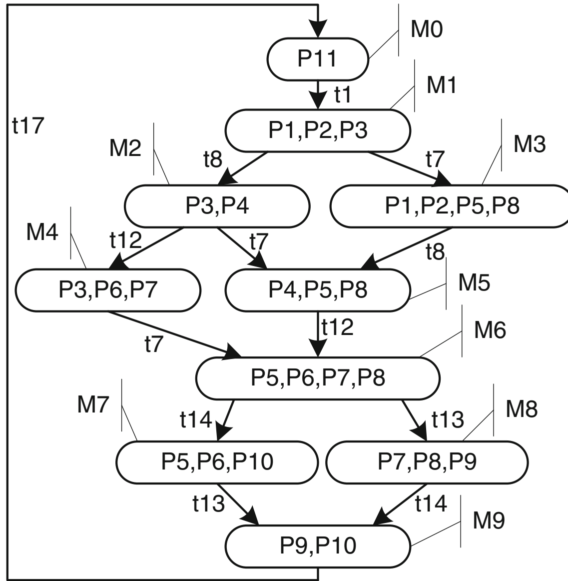


Fig. 2 Reachability graph of the Petri net

Distribution of the Petri net tokens among places describes the current global state M. New marking M, after firing of any enabled transition, is the next global state $@M$. From the current global state M, the modeled controller goes to the next internal global state $@M$, generating the registered $@y$ output signals.

In order to determine the SM-subnets covering of the Petri nets, formal reasoning using propositional Gentzen calculus was used [1, 8, 10–12]. As a result of local and global state space analysis, state machine subnets are obtained. The subnets are subsequently marked with different colors. The subnets are accordingly mapped to tokens, places, transitions and input and output signals.

Hypergraph of concurrency (Fig. 3) represents all the global states of a discrete system described by the Petri net. Its hyperedges relate concurrent places belonging to the same global states and correspond to the respective vertices of reachability graph (Fig. 2).

Hypergraph of non-concurrency, or sequentiality (Fig. 4), is a complement of the hypergraph of concurrency. Its hyperedges correspond to sets of sequential Petri net places, where only one place can be marked at the same time, from state machine subnets. They can be calculated as the exact transversals of concurrency hypergraph (Fig. 3) [11, 14].

$$\vdash P_{11}, (P_1 \ or \ P_2 \ or \ P_3), (P_3 \ or \ P_4), \\ (P_1 \ or \ P_2 \ or \ P_5 \ or \ P_8), \ldots, (P_9 \ or \ P_{10})$$

The full hypergraph of sequentiality shown in Fig. 4 contains six hyperedges $\{i_1 - i_6\}$. There are two net covers with exactly four hyperedges: $\{i_3, i_4, i_1, i_2\}$,

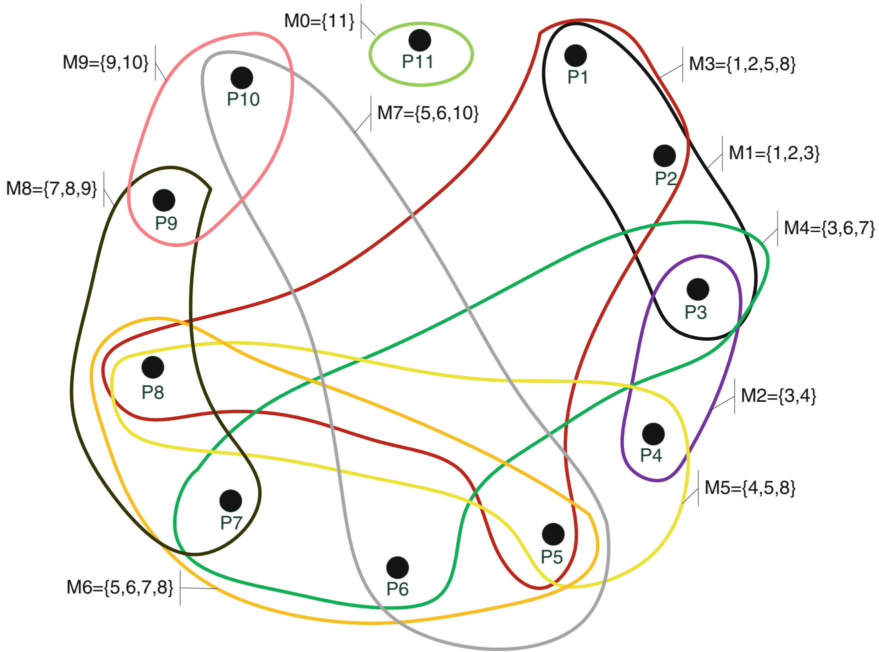


Fig. 3 Hypergraph of concurrency

$\{i_3, i_4, i_5, i_6\}$. Both covers contain essential hyperedges i_3 and i_4, which are the only ones to cover places P_5 and P_8. The first four sequential subnets $i_1 - i_4$ are sufficient to cover all places of the net (Fig. 5).

It should be noted that the sequentiality graph hyperedges correspond to the invariants which can be obtained using integer linear programming (ILP) methods. Alternatively they can be read from the marking reachability graph Fig. 2, as transversals of subsets of places marked in parallel [5, 6].

$\vdash P_1, P_4, \mathbf{P_6}, \mathbf{P_8}, P_{10}, P_{11};$
$\vdash P_1, P_4, P_6, P_9, P_{11};$ (i_1)
$\vdash P_1, \mathbf{P_4}, \mathbf{P_5}, P_7, P_9, P_{11};$
$\vdash P_1, P_4, P_7, P_{10}, P_{11};$ (i_6)
$\vdash P_2, P_4, \mathbf{P_6}, \mathbf{P_8}, P_{10}, P_{11};$
$\vdash P_2, P_4, P_6, P_9, P_{11};$ (i_5)
$\vdash P_2, \mathbf{P_4}, \mathbf{P_5}, P_7, P_9, P_{11};$
$\vdash P_2, P_4, P_7, P_{10}, P_{11};$ (i_2)
$\vdash P_3, \mathbf{P_5}, \mathbf{P_7}, P_{10}, P_{11};$
$\vdash P_3, P_5, P_9, P_{11};$ (i_4)
$\vdash \mathbf{P_3}, \mathbf{P_6}, P_8, P_9, P_{11};$
$\vdash P_3, P_8, P_10, P_{11};$ (i_3)

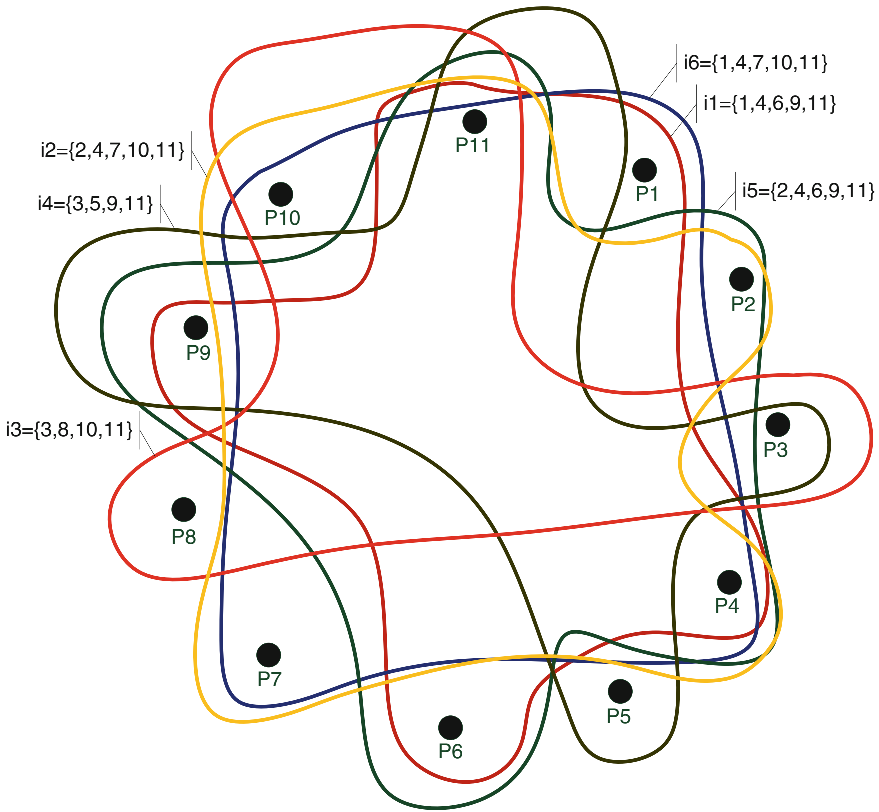


Fig. 4 Hypergraph of sequentiality

Finally selected cover of Petri net by state machine components could be represented only by four transversals of the net state space (Fig. 3). It is described as a reduced hypergraph of sequentiality (Fig. 5):

$$\vdash \; P_1, P_4, P_6, P_9, P_{11}$$
$$\vdash \; P_2, P_4, P_7, P_{10}, P_{11}$$
$$\vdash \; P_3, P_8, P_{10}, P_{11}$$
$$\vdash \; P_3, P_5, P_9, P_{11}$$

4 Coloring with the Use of Siphons and Traps

What differentiates sequent calculus from other methods, e.g. those used in [2, 8, 10], is that the conversion to clausal form is not required. Moreover, by using cut and consensus a laborious process of results selection is avoided. Siphons and traps that are not minimal are eliminated beforehand.

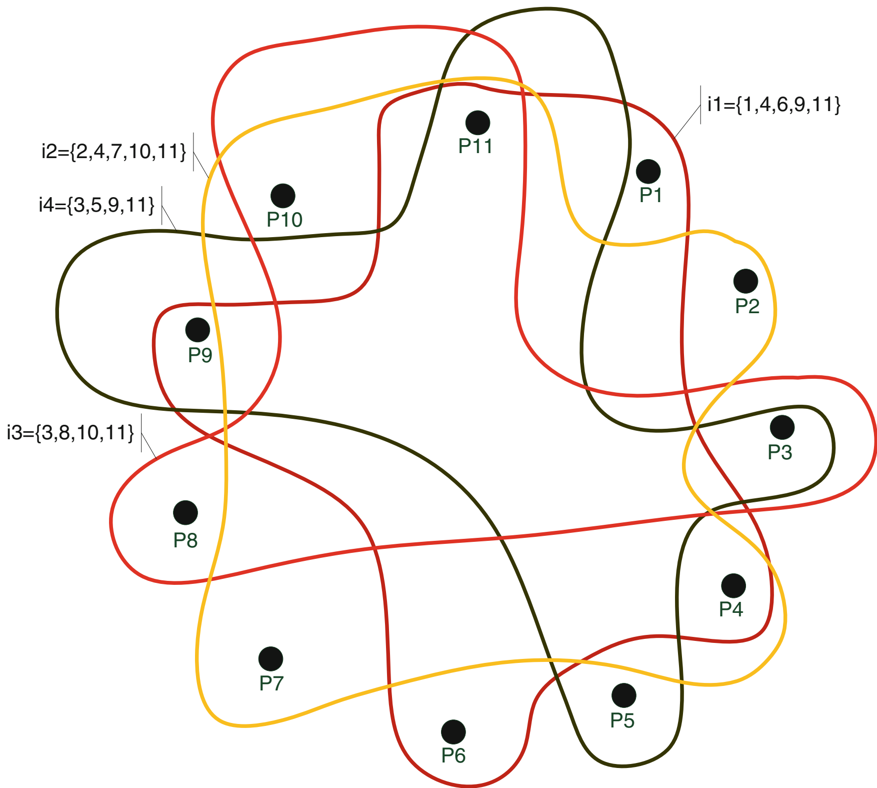


Fig. 5 Cover of the reduced sequentiality hypergraph

Table 1 Group sequents of traps and siphons

Transitions	Traps sequent	Siphons sequent
t_1	$(P_{11} \rightarrow (P1 + P2 + P3))$	$((P_1 + P_2 + P_3) \rightarrow P_{11})$
t_8	$((P_1 + P_2) \rightarrow P_4)$	$(P_4 \rightarrow (P_1 + P_2))$
t_7	$(P_3 \rightarrow (P_5 + P_8))$	$((P_5 + P_8) \rightarrow P_3)$
t_{12}	$(P_4 \rightarrow (P_6 + P_7))$	$((P_6 + P_7) \rightarrow P_4)$
t_{13}	$((P_5 + P_6) \rightarrow P_9)$	$(P_9 \rightarrow (P_5 + P_6))$
$t14$	$((P_7 + P_8) \rightarrow P_{10})$	$(P_{10} \rightarrow (P_7 + P_8))$
$t17$	$((P_9 + P_{10}) \rightarrow P_{11}) \vdash$	$(P_{11} \rightarrow (P_9 + P_{10})) \vdash$

The Petri net will be analysed in the next steps. Gentzen sequents, showing in a symbolic way all the relations of direct transition between input and output places of all transitions, are determined on the basis of the topological structure of an uninterpreted Petri net. Using the method presented in [8, 10] separated sequents of siphons and traps were created (Table 1).

Table 2 Traps and siphons

	Traps	Siphons	Color
i_1	$P_1, P_4, P_6, P_9, P_{11}$	$P_1, P_4, P_6, P_9, P_{11}$	C_3
i_6	$P_1, P_4, P_7, P_{10}, P_{11}$	$P_1, P_4, P_7, P_{10}, P_{11}$	C_5
i_5	$P_2, P_4, P_6, P_9, P_{11}$	$P_2, P_4, P_6, P_9, P_{11}$	C_6
i_2	$P_2, P_4, P_7, P_{10}, P_{11}$	$P_2, P_4, P_7, P_{10}, P_{11}$	C_4
i_4	P_3, P_5, P_9, P_{11}	P_3, P_5, P_9, P_{11}	C_2
i_3	P_3, P_8, P_{10}, P_{11}	P_3, P_8, P_{10}, P_{11}	C_1

In order to check whether the Petri net is live, traps equal to siphons (deadlocks) are calculated [1, 2, 9, 11, 12]. Sets of marked traps contained in siphons determine potential state machine subnet, present in the Petri net. Each of the subnets is marked with a different color, flagging also its places and transitions. Traps not equal to siphons indicate potential net defects. The net is not live, if not all the siphons contain traps. We suggest the following method of Petri net state space analysis. We use the net depicted in Fig. 1 as an example:

1. Using the rule-based symbolic description of the Petri net, create the group sequent of traps (Table 1).
2. Reduce the group sequent of traps to single normalized elementary sequents and remove their right sides (Table 2).
3. Remove the sequents of traps not containing a marked place symbol.
4. Using the rule-based symbolic description of the Petri net, create the group sequent of siphons (deadlocks) (Table 1).
5. Reduce the group sequent of siphons to elementary sequents and remove their right sides. Apply the consensus rule to the previously selected sequents of traps in every chosen siphon. If the considered sequent becomes dominated, go to step 8 (all sequents are dominated).
6. The traps which are equal to siphons determine covering of sequentiality hypergraph by the hyperegdes corresponding to potential state machine subnets (in the considered example colors $C_1 - C_6$ are assigned to the hyperedges—Table 2).
7. Find the minimal covers of the sequentiality hypergraph (for the considered example these are $\{C_1, C_2, C_3, C_4\}$ or $\{C_1, C_2, C_5, C_6\}$); go to step 9.
8. The Petri net is not live (not the case of Petri net in Fig. 1).
9. End.

After removing the right sides of the reduced sequents and after leaving only the siphons dominated by marked traps, we obtained six potential automata subnets. These sets correspond to the hyperedges of the sequentiality hypergraph $\{i_1, \ldots, i_6\}$ (Fig. 2). Following step 5 in the above algorithm, we conclude that the Petri net is live, since each edge of the hypergraph contains a marked trap. Subsets determining traps and siphons are pairwise identical. The cover (i_3, i_4, i_1, i_2) was chosen for Petri net encoding. Subnets were assigned colors according to Table 2. These colors were placed on the net shown in Fig. 1.

5 Conclusion

One of the major advantages of the proposed method is the localization of a potential defects. When some siphons are not equal to traps then net is not live. These siphons directly indicate the location of the defects.

Proposed methods discover all possible colorings and for most cases the complexity and time of execution is not acceptable for application in design tools. These methods are typically used for the purpose of verification of heuristic methods.

In addition, the proposed methods are suitable also for hierarchical specifications of logic controllers. In this case hierarchical specification of the logic controller should be prepared in the form of a Petri macronet.

References

1. Adamski, M., & Tkacz, J. (2012). Formal reasoning in logic design of reconfigurable controllers. In *Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems PDeS'12* (pp. 1–6). Brno, Czech Republic.
2. Adamski, M., Karatkevich, A., & Węgrzyn, M. (eds.) (2005). *Design of embeded control systems*. New York: Springer Science+Business Media Inc.
3. Ben-Ari, M. (2012). *Mathematical logic for computer science* (3rd ed.). London: Springer.
4. Biliński, K., Adamski, M., Saul, J., & Dagless, E. (1994). Petri-net-based algorithms for parallel-controller synthesis. *IEE Proceedings – Computers and Digital Techniques*, *141*(6), 405–412.
5. Gallier, J. H. (1985). *Logic for computer science: Foundations of automatic theorem proving*. New York: Harper & Row Publishers.
6. Girault, C., & Valk, R. (2003). *Petri nets for system engineering: A guide to modeling, verification, and applications*. Berlin: Springer.
7. Jensen, K., Kristensen, K., & Wells, L. (2007). Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, *9*(3), 213–254.
8. Karatkevich, A. (2007). *Dynamic analysis of Petri net-based discrete systems* (Vol. 356). Lecture notes in control and information sciences. Berlin: Springer.
9. Kozłowski, T., Dagless, E., Saul, J., Adamski, M., & Szajna, J. (1995). Parallel controller synthesis using Petri nets. *IEE Proceedings – Computers and Digital Techniques*, *142*(4), 263–271.
10. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.
11. Tkacz, J. (2007). State machine type colouring of Petri net by means of using a symbolic deduction method. *Measurement Automation and Monitoring*, *53*(5), 120–122.
12. Tkacz, J., & Adamski, M. (2011). Calculation of state machine cover of safe Petri net by means of computer based reasoning. *Measurement Automation and Monitoring*, *57*(11), 1397–1400.
13. Węgrzyn, M., Wolański, P., Adamski, M., & Monteiro, J. (1997). Coloured Petri net model of application specific logic controller programs. In *Proceedings of IEEE International Symposium on Industrial Electronics ISIE'97* (Vol. 1, pp. 158–163). Guimarães, Portugal. Piscataway.
14. Wiśniewska, M., Wiśniewski, R., & Adamski, M. (2007). Usage of hypergraph theory in decomposition of concurrent automata. *Measurement Automation and Monitoring*, *53*(7), 66–68.

Modular Synthesis of Petri Nets

Jacek Tkacz and Marian Adamski

Abstract The chapter is concentrated on behavioral and structural specification of reconfigurable logic controllers (RLC). The initial description is given as a hierarchical modular control interpreted Petri net. On the abstract level of the logic synthesis a specification is written in formal propositional Gentzen sequent language. Rapid modeling in FPGA can be done directly from rule-based expressions, written in a hardware description language, for example in VHDL.

Keywords Sequents · Gentzen logic · Petri net · Logic synthesis · Logic controllers

1 Introduction

The chapter covers logic design techniques, which can be used for rigorous computer-based synthesis of Reconfigurable Logic Controllers (RLC) [1, 2, 7, 8]. Initially, the behavior of the controller is specified as hierarchical colored control interpreted Petri net. The decision rules, written in Propositional Sequent Logic, describe both structure of the net as well as the intended behavior of the logic controller, given in a professional hardware description language. Gentzen symbolic reasoning [10] establishes a link between Boolean expressions, commonly used in a digital system design on RTL level and the Petri net model of a Hierarchical Concurrent State Machine [3].

J. Tkacz (✉) · M. Adamski
Institute of Computer Engineering and Electronics,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: j.tkacz@iie.uz.zgora.pl

M. Adamski
e-mail: m.adamski@iie.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_7

The strategy developed and promoted in this chapter is based on the hierarchical decomposition of Petri nets into nested, self-contained and structurally ordered subnets, which are suitable for distributed state encoding as well as flexible reconfiguration. All structured modules are easily recognized by their symbolic names of the configuration (coordination) places, which are only marked if selected modules are active.

The proposed design methodology is based on a formal mapping of specification in propositional Gentzen logic into very close equivalent description of the implementation, which is accepted directly by VHDL [5].

The main goal of the proposed rapid design style is to preserve the self-evident correspondence among modular interpreted Petri net, the related symbolic rule-based specification, and final logic design expressions, which are directly mapped into configurable logic arrays FPGA [2, 16, 17].

2 Example of Control System

The controlled plant (Fig. 1) consists of three feeders, scales and content mixer. The example first introduced by P. Misiurewicz has been used as benchmark in several papers, among others in [5, 12].

The logic controller has six inputs $\{XN_1, XN_2, XF_1, XF_2, XF_3, XF_4\}$ and six outputs $\{YT_1, YT_2, YV_1, YN_2, YV_3, YM\}$. The example of control interpreted Petri net, which describes the behavior of control system is presented in Fig. 2. The Petri net places $\{P_1, P_2, \ldots, P_{11}\}$ stand for the local states of concurrent state machine. The transitions $t_1 \ldots t_9$ describe events in terms of local changes inside the Petri net state space. Boolean expressions $XN_1 \ldots XF_4$ called guards give the external conditions for transitions to be enabled and fired. The colored coordination places P_{12} and P_{13} in Fig. 2 are optional.

The guarded events are strongly related with transitions of the net (Table 1). The Moore type outputs $YT_1 \ldots YM$ are attached to places (Table 2). The basic,

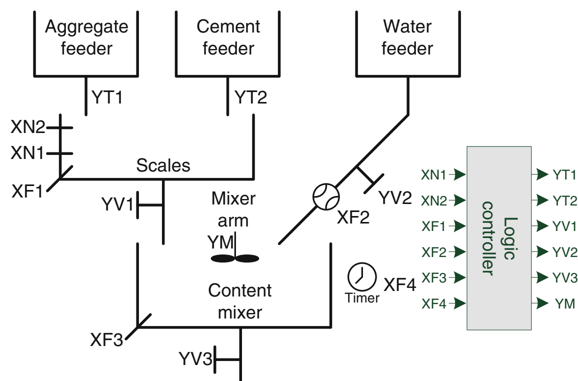


Fig. 1 Mechanical part of discrete control system

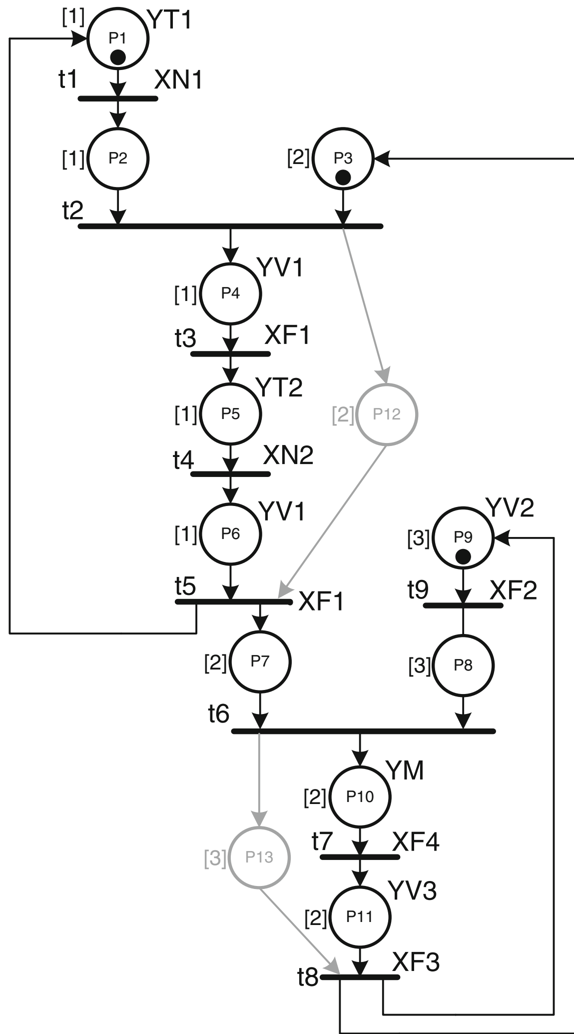


Fig. 2 Petri net model

one-level net could be colored by a designer during the initial specification process to demonstrate its preferable State Machines subnets (Fig. 2). These colors help to validate intuitively and formally the consistency of all sequential processes in the considered discrete state model. The control Petri net is covered by three separated State-Machine components SM_1, SM_2, SM_3, recognized by colors {[1], [2], [3]}:

$$SM_1[1] = \{P_1, P_2, P_4, P_5, P_6\};$$
$$SM_2[2] = \{P_3, P_{12}, P_7, P_{10}, P_{11}\};$$
$$SM_3[3] = \{P_9, P_8, P_{13}\};$$

Table 1 List of transitions and guards

Transition	Guard	Interpretation of guard
t_1	XN1	Required value of aggregate is reached
t_2	1	Always true
t_3	XF1	The scale is empty
t_4	XN2	Required value of cement is reached
t_5	XF1	The scale is empty
t_6	1	Always true
t_7	XF4	Ingredients are intermixed
t_8	XF3	Cement mixer is empty
t_9	XF2	Required value of cement is reached

Table 2 List of places and outputs

Place	Output	Interpretation of place
P_1	YT1	First dozing of cement
P_2	–	Waiting
P_3	–	Waiting
P_4	YV1	First emptying the scale
P_5	YT1	Second dozing of cement
P_6	YV1	Second emptying the scale
P_7	–	Waiting
P_8	–	Waiting
P_9	YV2	Dosing of water
P_{10}	YM	Mixing of compounds
P_{11}	YV3	Emptying the mixer

3 Place Centered Specification of Petri Net in Gentzen Logic

In formal description of the Petri net, letters stand for the symbols from Gentzen propositional logic [10, 11]. Symbol *and* denotes conjunction symbol *or* denotes disjunction, symbol *not*—negation, symbol $\leftarrow$ backward implication, symbol *xor*—exclusive or, symbol $\leftrightarrow$ equivalence.

The specification of Petri net is concentrated around places with their input and output transitions. Such strategy makes possible to represent Petri net places as a separated elementary parts of Petri net.

The logic description describes the changes of Petri net markings, separately for any place. The autonomous place P_n is considered together with its input $\{t_i \ldots t_j\}$ and output transitions $\{t_k \ldots t_l\}$ as a basic component of the net (Fig. 3). The precondition for firing transition t_i is: $t_i \Leftarrow p_m$ *and* p_l *and* $guard_{ti}$. The place safely gets its token

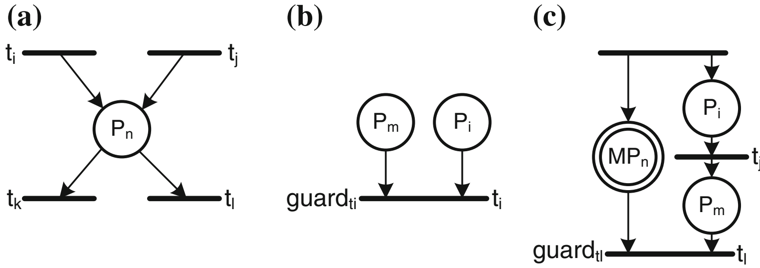


Fig. 3 Symbolic representation of Petri net parts

if one of its input transition fires. The next marking for a place P_n (Fig. 3a) is defined as follows:

$$@P_n \Leftarrow P_n \; xor \; ((t_i \; xor \; t_j) \; xor \; (t_k \; xor \; t_l)) \quad (1)$$

If net is safe *xor* operator in expression $(t_i \; xor \; t_j)$ can be replaced by *or*. If the place deterministically loses its token the expression $(t_k \; xor \; t_l)$ is simplified to $(t_k \; or \; t_l)$ [4]. The precondition of local transitions t_i (Fig. 3b) is defined as follows:

$$t_i \Leftarrow P_m \; and \; P_i \; and \; guard_{ti} \quad (2)$$

Macroplace MP_n from the example (Fig. 3c) contains sequential places P_i and P_m. The common, boundary transition t_l presented on Fig. 3c can be described as follows:

$$t_l \Leftarrow MP_n \; and \; P_m \; and \; guard_{tl} \quad (3)$$

To make the specification close with VHDL syntax and semantics, the sequents with empty left side are used: "$\vdash \Phi$;" where Φ is formula in propositional logic. Symbol @ defines *next* operator from propositional temporal logic and it is usually also omitted.

The first part of state-event-state (place-transition-place) description of one level Petri net is given in the Table 3.

Each line of description represents a single event in Petri net as a transition with its preconditions.

The Moore type combinational outputs are related with places, which contain token when that output is active:

$$\vdash YT_1 \Leftarrow P_1;$$
$$\vdash YV_1 \Leftarrow P_4 \; or \; P_6;$$
$$\ldots$$
$$\vdash YV_3 \Leftarrow P_{11};$$

Table 3 Precondition and outputs

Precondition of transitions	Moore type outputs
$\vdash t_1 \Leftarrow P_1$ *and* XN_1	$\vdash YT_1 \Leftarrow P_1$
$\vdash t_2 \Leftarrow P_2$ *and* P_3	$\vdash YV_1 \Leftarrow P_4$
$\vdash t_3 \Leftarrow P_4$ *and* XF_1	$\vdash YT_2 \Leftarrow P_5$
$\vdash t_4 \Leftarrow P_5$ *and* XN_2	$\vdash YV_1 \Leftarrow P_6$
$\vdash t_5 \Leftarrow P_6$ *and* XF_1	$\vdash YV_2 \Leftarrow P_9$
$\vdash t_6 \Leftarrow P_7$ *and* P_8	$\vdash YV_3 \Leftarrow P_{11}$
$\vdash t_7 \Leftarrow P_{10}$ *and* XF_4	$\vdash YVM \Leftarrow P_{10}$
$\vdash t_8 \Leftarrow P_{11}$ *and* XF_3	
$\vdash t_9 \Leftarrow P_9$ *and* XF_2	

Different forms of rule-based specification can be found in papers [1, 4, 14]. Petri net specification format (PNSF) serves as convenient textual form for an automatic translation of transition rules into VHDL or Verilog code [5].

After concurrent one-hot encoding symbols $P_1 \ldots P_{11}$ are treated as names of flip-flops contained in distributed local state register. The outputs can be traditionally generated in combinational circuit: $YT_1 \Leftarrow P_1 \ldots YM \Leftarrow P_{10}$ (Table 3). Finally, changes of place markings:

$$
\begin{aligned}
&\vdash @P_1 &&\Leftarrow P_1 \ xor \ (t_5 \ xor \ t_1);\\
&\vdash @P_2 &&\Leftarrow P_2 \ xor \ (t_1 \ xor \ t_2);\\
&\vdash @P_3 &&\Leftarrow P_3 \ xor \ (t_8 \ xor \ t_2);\\
&\vdash @P_4 &&\Leftarrow P_4 \ xor \ (t_2 \ xor \ t_3);\\
&\vdash @P_5 &&\Leftarrow P_5 \ xor \ (t_3 \ xor \ t_4);\\
&\vdash @P_6 &&\Leftarrow P_6 \ xor \ (t_4 \ xor \ t_5);\\
&\vdash @P_7 &&\Leftarrow P_7 \ xor \ (t_5 \ xor \ t_6);\\
&\vdash @P_8 &&\Leftarrow P_8 \ xor \ (t_9 \ xor \ t_6);\\
&\vdash @P_9 &&\Leftarrow P_9 \ xor \ (t_8 \ xor \ t_9);\\
&\vdash @P_{10} &&\Leftarrow P_{10} \ xor \ (t_6 \ xor \ t_7);\\
&\vdash @P_{11} &&\Leftarrow P_{11} \ xor \ (t_7 \ xor \ t_8);
\end{aligned}
$$

Some of registered outputs can serve also as local one-hot state codes, except YV_1, which is generated twice in local states P_4 and P_6: $YV_1 \Leftarrow P_4 \ or \ P_6$:

$$
\begin{aligned}
&\vdash @YT_1 &&\Leftarrow YT_1 \ xor \ (t_5 \ xor \ t_1);\\
&\vdash @YT_2 &&\Leftarrow YT_2 \ xor \ (t_3 \ xor \ t_4);\\
&\vdash @YV_2 &&\Leftarrow YV_2 \ xor \ (t_8 \ xor \ t_9);\\
&\vdash @YVM &&\Leftarrow YVM \ xor \ (t_6 \ xor \ t_7);\\
&\vdash @YV_3 &&\Leftarrow YV_3 \ xor \ (t_7 \ xor \ t_8);
\end{aligned}
$$

The full list of preconditions of local transitions and descriptions of Moore type outputs is given in Table 3.

Table 4 Example of encoding

$SM_1[1]\{Q_1, Q_2, Q_3\}$	$SM_2[2]\{Q_4, Q_5, Q_6\}$	$SM_3[3]\{Q_7, Q_8\}$
$P_1 = 000$	$P_3 = 000$	$P_9 = 00$
$P_2 = 001$	$P_{12} = 001$	$P_8 = 01$
$P_4 = 011$	$P_7 = 011$	$P_{13} = 11$
$P_5 = 010$	$P_{10} = 010$	
$P_6 = 110$	$P_{11} = 110$	

4 Encoding Inside State Machine Modules

Encoding from Table 4 can be used for more compact dense state space. The number of logic variables is reduced to eight by commercial tools [16]. Every state machine subnet is encoded separately. The changes of marking are now represented by changes of local state variables. The current value of a registered signal is presented as Q, but its next value is written as $@Q$ [1].

$$\begin{aligned}
&--\,SM_1\,--\\
&\vdash @Q_1 \Leftarrow Q_1\ xor\ (t_4\ xor\ t_5);\\
&\vdash @Q_2 \Leftarrow Q_2\ xor\ (t_2\ xor\ t_5);\\
&\vdash @Q_3 \Leftarrow Q_3\ xor\ (t_1\ xor\ t_3);\\
&--\,SM_2\,--\\
&\vdash @Q_4 \Leftarrow Q_4\ xor\ (t_7\ xor\ t_8);\\
&\vdash @Q_5 \Leftarrow Q_5\ xor\ (t_5\ xor\ t_8);\\
&\vdash @Q_6 \Leftarrow Q_6\ xor\ (t_2\ xor\ t_6);\\
&--\,SM_3\,--\\
&\vdash @Q_7 \Leftarrow Q_7\ xor\ (t_6\ xor\ t_8);\\
&\vdash @Q_8 \Leftarrow Q_8\ xor\ (t_9\ xor\ t_8);
\end{aligned}$$

In this case combinational outputs should be generated as follows:

$$\begin{aligned}
&\vdash YT_1 \Leftarrow not\ Q_1\ and\ not\ Q_2\ and\ not\ Q_3;\\
&\dots\\
&\vdash YV_3 \Leftarrow Q_4\ and\ Q_5\ and\ not\ Q_6;
\end{aligned}$$

5 Implementation of Colored Hierarchical Macronet

5.1 Modular Specification of Logic Controller

Together with coloring a Petri net can be converted into suitable hierarchical description. As an example, the initial, basic net from Fig. 2 was reduced to the macronet with macroplaces $MP_1 \dots MP_6$ (Fig. 4). Transitions with more than one input place

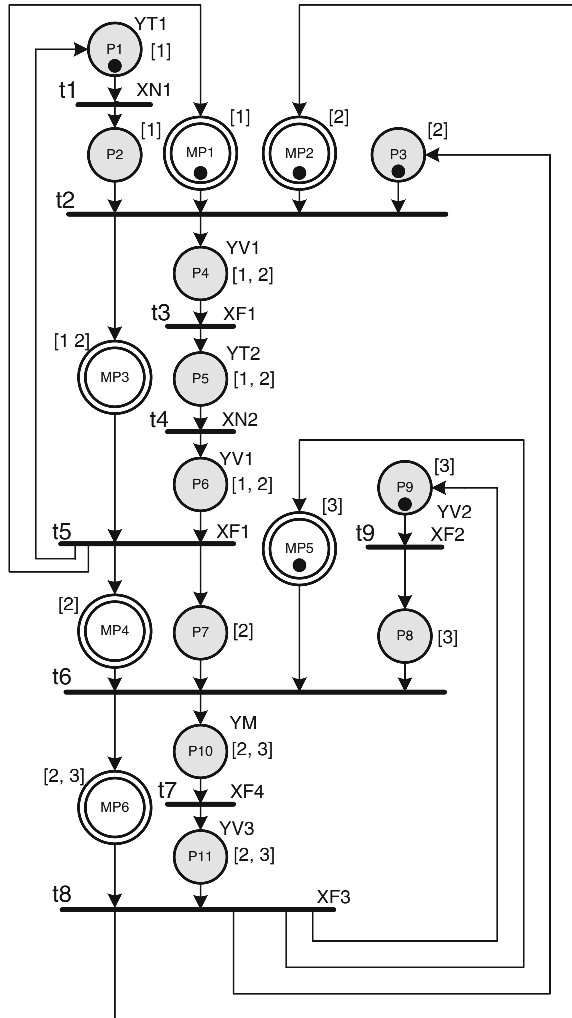


Fig. 4 Hierarchical macronet

or more than one output place, such as t_2, t_5, t_6, t_8 are called boundary transitions. Transfer transitions with one input and one output places are hidden inside first order macroplaces. Fusion of Series Places (FSP) and fusion of Parallel Places (FPP) [11, 14] are used recursively during coloring [6, 15], until the macronet becomes irreducible.

It should be noted, that the macroplaces which are painted with disjoint set of colors are evidently concurrent to each other. The macroplaces sharing the same color are sequentially related to each other. The special implicit configuration (coordination) places $MP_1 \ldots MP_6$ detect all the Petri net subnets, which they dominate. During the hierarchical state encoding only a proper subset of them is necessary to detect the groups of places [3].

5.2 General Template for Modular Logic Design

Symbols $MP_1 \ldots MP_6$ are the names of macroplaces as well as names of their coordination places (subnet flags) from Fig. 4. Petri net places $p_1 \ldots p_{11}$ are related to capital letters denoting single bit memory elements—flip-flops: $P_1 \ldots P_{11}$.

The main part of a novel template of formal description of Petri net in Gentzen sequent logic language is as follows:

Preconditions of boundary transitions:

$$\begin{aligned}
&\vdash t_2 \Leftarrow MP_1 \textit{ and } MP_2 \textit{ and } P_2;\\
&\vdash t_5 \Leftarrow MP_3 \textit{ and } P_6;\\
&\vdash t_6 \Leftarrow MP_4 \textit{ and } MP_5 \textit{ and } P_7 \textit{ and } P_8;\\
&\vdash t_8 \Leftarrow MP_6 \textit{ and } P_{11};
\end{aligned}$$

Precondition of local transitions:

$$\begin{aligned}
&\vdash t_1 \Leftarrow MP_1 \textit{ and } P_1 \textit{ and } XN_1;\\
&\vdash t_3 \Leftarrow MP_3 \textit{ and } P_4 \textit{ and } XF_1;\\
&\vdash t_4 \Leftarrow MP_3 \textit{ and } P_5 \textit{ and } XN_2;\\
&\vdash t_7 \Leftarrow MP_6 \textit{ and } P_{10} \textit{ and } XF_4;\\
&\vdash t_9 \Leftarrow MP_5 \textit{ and } P_9 \textit{ and } XF_2;
\end{aligned}$$

Flags of macrostates (macroplaces):

$$\begin{aligned}
&\vdash @MP_1 \Leftarrow MP_1 \textit{ xor } (t_5 \textit{ xor } t_2);\\
&\vdash @MP_2 \Leftarrow MP_2 \textit{ xor } (t_8 \textit{ xor } t_2);\\
&\vdash @MP_3 \Leftarrow MP_3 \textit{ xor } (t_2 \textit{ xor } t_5);\\
&\vdash @MP_4 \Leftarrow MP_4 \textit{ xor } (t_5 \textit{ xor } t_6);\\
&\vdash @MP_5 \Leftarrow MP_5 \textit{ xor } (t_8 \textit{ xor } t_6);\\
&\vdash @MP_6 \Leftarrow MP_6 \textit{ xor } (t_6 \textit{ xor } t_8);
\end{aligned}$$

Places are encoded inside macroplaces. Changes of local places are as follows:

$$\begin{aligned}
&\vdash @P_1 \Leftarrow (P_1 \textit{ and } MP_1) \textit{ xor } (t_5 \textit{ xor } t_1);\\
&\vdash @P_2 \Leftarrow (P_2 \textit{ and } MP_1) \textit{ xor } (t_1 \textit{ xor } t_2);\\
&\ldots\\
&\vdash @P_{10} \Leftarrow (P_{10} \textit{ and } MP_6) \textit{ xor } (t_6 \textit{ xor } t_7);\\
&\vdash @P_{11} \Leftarrow (P_{11} \textit{ and } MP_6) \textit{ xor } (t_7 \textit{ xor } t_8);
\end{aligned}$$

5.3 One-Hot Encoding of Macroplaces

For a rapid prototyping, macroplaces are coded by means of registered outputs $Q_1 \ldots Q_6$ as shown in Fig. 5:

Fig. 5 Sample VHDL code

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity reactor is
port (CLK, RESET: in std_logic;
      XN1, XN2, XF1, XF2, XF3, XF4: in std_logic;
      YT1, YT2, YV1, YV2, YV3, YM: out std_logic);
end reactor;

architecture reactor_t of reactor is
   signal Q1, Q2, Q3, Q4, Q5, Q6: std_logic;
   signal Y_T1, Y_T2, Y_V1a, Y_V1b: std_logic;
   signal Y_V2, Y_V3, Y_M: std_logic;
   signal t1, t2, t3, t4, t5, t6, t7, t8, t9: std_logic;
begin
   --Precondition of border transitions
   t2 <= Q1 and Q2 and not Y_T1 and not Y_V3;
   t5 <= Q3 and Y_V1b and XF1;
   t6 <= Q4 and Q5 and not Y_V1b and not Y_V2;
   t8 <= Q6 and Y_V3 and XF3;
   --Precondition of local transitions
   t1 <= Q1 and Y_T1 and XN1;
   t3 <= Q3 and Y_V1a and XF1;
   t4 <= Q3 and Y_T2 and XN2;
   t7 <= Q6 and Y_M and XF4;
   t9 <= Q5 and Y_V2 and XF2;

   FF:process (CLK,RESET) -- Transition firing
   begin
     if RESET = '1' then  -- Asynchronous reset
            Q1 <= '1'; Q2 <= '1'; Q3 <= '0'; Q4 <= '0';
            Q5 <= '1'; Q6 <= '0';
            Y_T1 <= '1'; Y_T2 <= '0'; Y_V1a <= '0';
            Y_V1b <= '0'; Y_V2 <= '1';
            Y_V3 <= '0'; Y_M <= '0';
     elseif rising_edge(CLK) then

            Q1 <= Q1 xor (t5 xor t2);
            Q2 <= Q2 xor (t8 xor t2);
            Q3 <= Q3 xor (t2 xor t5);
            Q4 <= Q4 xor (t5 xor t6);
            Q5 <= Q5 xor (t8 xor t6);
            Q6 <= Q6 xor (t6 xor t8);
            Y_T1 <= (Y_T1 and Q1) xor (t5 xor t1);
            Y_T2 <= (Y_T2 and Q3) xor (t3 xor t4);
            Y_V1a <= (Y_V1a and Q3) xor (t2 xor t3);
            Y_V1b <= (Y_V1b and Q3) xor (t4 xor t5);
            Y_V2 <= (Y_V2 and Q5) xor (t8 xor t9);
            Y_V3 <= (Y_V3 and Q6) xor (t7 xor t8);
            Y_M  <= (Y_M and Q6) xor (t6 xor t7);
     end if;
   end process;
   -- Outputs
   YT1 <= Y_T1;
   YT2 <= Y_T2;
   YV1 <= Y_V1a or Y_V1b;
   YV2 <= Y_V2;
   YV3 <= Y_V3;
   YM  <= Y_M;
end reactor_t;
```

$$MP_1 \Leftrightarrow Q_1;\ MP_2 \Leftrightarrow Q_2;\ MP_3 \Leftrightarrow Q_3;$$
$$MP_4 \Leftrightarrow Q_4;\ MP_5 \Leftrightarrow Q_5;\ MP_6 \Leftrightarrow Q_6;$$

It is easy, but not recommended, to find codes for local places P_1 and P_{11}, using only additional variables $Q_7 \ldots Q_{17}$.

5.4 Local Encoding Inside Macroplaces with Registered Outputs

The local places can be encoded using registered outputs. The codes of local places are as follows:

$$\begin{array}{lll} P_1 \Leftrightarrow YT_1; & P2 \Leftrightarrow not\ YT_1; & P3 \Leftrightarrow not\ YV_3; \\ P_4 \Leftrightarrow YV_{1a}; & P5 \Leftrightarrow YT_2; & P6 \Leftrightarrow YV_{1b}; \\ P_7 \Leftrightarrow not\ YV_1; & P8 \Leftrightarrow not\ YV_2; & P9 \Leftrightarrow YV_2; \\ P_{10} \Leftrightarrow YM; & P_{11} \Leftrightarrow YV_3; & \end{array}$$

After that kind of encoding of places $\{P_1, P_4, P_5, P_9, P_{10}\}$ the preconditions of local transitions are as:

$$\begin{array}{l} \vdash t_1 \Leftarrow MP_1\ and\ YT_1\ and\ XN_1; \\ \vdash t_3 \Leftarrow MP_3\ and\ YV_{1a}\ and\ XF_1; \\ \vdash t_4 \Leftarrow MP_3\ and\ YT_2\ and\ XN_2; \\ \vdash t_7 \Leftarrow MP_6\ and\ YM\ and\ XF_4; \\ \vdash t_9 \Leftarrow MP_5\ and\ YV_2\ and\ XF_2; \end{array}$$

As a result of replacing the names of places by related output names, the changes of local places are described as follows:

$$\begin{array}{ll} \vdash @YT_1 \Leftarrow (YT_1\ and\ MP_1)\ xor\ (t_5\ xor\ t_1); & /*\ @P_1\ */ \\ \vdash @YV_{1a} \Leftarrow (YV_{1a}\ and\ MP_3)\ xor\ (t_2\ xor\ t_3); & /*\ @P_4\ */ \\ \vdash @YT_2 \Leftarrow (YT_2\ and\ MP_3)\ xor\ (t_3\ xor\ t_4); & /*\ @P_5\ */ \\ \vdash @YV_{1b} \Leftarrow (YV_{1b}\ and\ MP_3)\ xor\ (t_4\ xor\ t_5); & /*\ @P_6\ */ \\ \vdash @YV_2 \Leftarrow (YV_2\ and\ MP_5)\ xor\ (t_8\ xor\ t_9); & /*\ @P_9\ */ \\ \vdash @YM \Leftarrow (YM\ and\ MP_6)\ xor\ (t_6\ xor\ t_7); & /*\ @P_{10}\ */ \\ \vdash @YV_3 \Leftarrow (YV_3\ and\ MP_6)\ xor\ (t_7\ xor\ t_8); & /*\ @P_{11}\ */ \end{array}$$

5.5 State Machine Style for Macroplace Encoding

As a next optimization the macroplaces can be encoded in state machine style. The first subnet corresponding to color [1], which contains macroplaces MP_1 and MP_3 can be encoded by one logic variable (Q_1). The second subnet, which

contains macroplaces MP_5 and MP_6 can be encoded also by one logic variable (Q_3). The last subnet corresponding to color [3], which contains macroplaces $\{MP_2, MP_3, MP_4, MP_6\}$ need two logic variable Q_2 and Q_4. Macroplaces MP_2 and MP_4 get one-hot codes:

$$\begin{array}{lll} MP_1 \Leftrightarrow Q_1; & MP_3 \Leftrightarrow not\ Q_1; & MP_5 \Leftrightarrow Q_3; \\ MP_6 \Leftrightarrow not\ Q_3; & MP_2 \Leftrightarrow Q_2; & MP_4 \Leftrightarrow Q_4; \end{array} \tag{4}$$

The symbol $\Leftrightarrow$ used above denotes logic equivalence. The number of flip flops is reduced from six to four. The number of expressions describing flags is now equal only four:

$$\begin{array}{l} \vdash @Q_1 \Leftarrow Q_1\ xor\ (t_5\ xor\ t_2); \\ \vdash @Q_2 \Leftarrow Q_2\ xor\ (t_8\ xor\ t_2); \\ \vdash @Q_4 \Leftarrow Q_4\ xor\ (t_5\ xor\ t_6); \\ \vdash @Q_3 \Leftarrow Q_3\ xor\ (t_8\ xor\ t_6); \end{array}$$

In this case the preconditions of boundary transitions and precondition of local transitions should be also changed by replacing macroplaces by codes presented in expressions (4).

6 VHDL-Style of the Modular Petri Net Description

The preferable way of controller rapid prototyping is hierarchical design from a formal assertion-based [9] behavioral description, using professional HDL syntax. One of the possible version of general template [5] is presented in Fig. 5.

For pragmatic reasons the controller is realized as a synchronous digital system with distributed state register $MP_1 \ldots MP_6$ and distributed output register $YT_1 \ldots YM$.

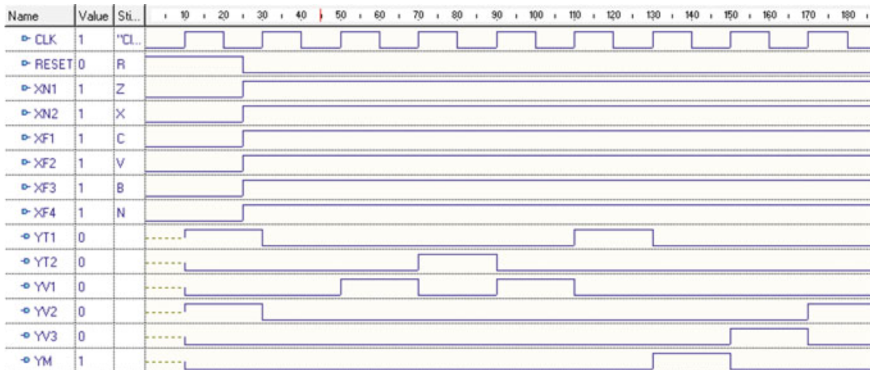


Fig. 6 Simulation results from AHDL tool

The state register and the output register can be merged. All concurrently enable transitions can fire independently, in any order. It is considered that after animation and classical analysis, the implemented interpreted Petri net is checked as safe, live, reversible and without conflicts, which are not solved [13, 14]. Anyway, if some transitions of the net would be in conflicts or the net is not safe, the detected partial state of the net is frozen (state changes stop). Registered outputs can be used both for precondition and local states coding. The simulation results obtained from Active-HDL tool is shown in Fig. 6.

7 Results of Experiments

The macroplace-centered and place-centered decomposition and encoding of SM-colored Petri net is preferable from FPGA resources utilization point of view [7, 16]. Additionally, it enables flexible reusing of previously tested, encoded Petri net components. Synthesis after classic hierarchical one-hot state encoding of macroplaces and places needs 17 flip-flops. In case of economical rapid concurrent one-hot local encoding of merged places and registered outputs (Fig. 5) it is necessary to use only 13 shared additional encoding variables. Synthesis result using Xilinx Vertex 2 Pro is: 12 slices, 13 flip-flops and 23 LUTs. After dense encoding of macroplaces the number of flip-flops is reduced to 11.

Hierarchical encoding using macroplaces and registered outputs gives balanced economical synthesis results as well flexibility during redesign of the controller. The coordination places serve also as flags during partial reconfiguration of the net. After modification of the water feeder from mechanical part (Fig. 1) it is easy to find local places P_8 and P_9, which are encapsulated in MP_5 and replace them by another subset without destroying the other parts of previous design.

The minimum number of coding variables for classic implementation with separated linked State Machine Components of the net is equal to eight. It is necessary to use complicated logic expressions for register excitation and decoding the seven outputs.

8 Summary

The rigorous digital design process starts from hierarchical concurrent state machine model (HCSM), which has been formally derived from modular, colored control interpreted Petri net. The colored tokens, arcs, places and transitions separate hierarchically and concurrently related State Machine components. The rule-based textual logic description of Petri net in VHDL syntax is accepted by professional design tools like Active-HDL (Aldec, USA) and Xlinx ISE. The flexible, readable template for Petri net description is directly recognized by VHDL compiler and simulator as well as by formal reasoning system. The logic specification is one-to-one mapped into

Field Programmable Gate Array macrocells. Combinatorial procedures in formal design of logic controller are supported by Gentzen sequent calculus.

The experimental design system can be used as a shell for existing proprietary tools, developed at Zielona Góra, as well with standard professional environment for digital synthesis and verification. It can also support the assertion-based design methodology of application specific logic controllers with checking techniques embedded in configurable hardware. The advantages are structured and modular state space of logic controllers, suitable for model checking. The self-evident VHDL template is suitable for rapid modifications also by hand.

References

1. Adamski, M. (2001). Specification and synthesis of Petri net based reprogrammable logic controller. In *Proceedings of 5th IFAC international conference on programmable devices and embedded systems PDeS'01* (pp. 95–100). Czech Republic: Brno.
2. Adamski, M. (2005). Formal logic design of reprogrammable controllers. In M. Adamski, A. Karatkevich, & M. Węgrzyn (Eds.), *Design of embedded control systems* (pp. 15–26). New York: Springer.
3. Adamski, M., & Tkacz, J. (2012). Formal reasoning in logic design of reconfigurable controllers. In *Proceedings of 11th IFAC/IEEE international conference on programmable devices and embedded systems PDeS'12* (pp. 1–6). Czech Republic: Brno.
4. Adamski, M., & Węgrzyn, M. (2009). Design of reconfigurable logic controllers from Petri net-based specifications. *4th IFAC workshop on discrete-event system design—DESDes'09* (pp. 233–238). Gandia Beach, Spain.
5. Adamski, M., & Węgrzyn, M. (2009). Petri nets mapping into reconfigurable logic controllers. *Electronics and Telecommunications Quarterly*, *55*(2), 157–182.
6. Biliński, K., Adamski, M., Saul, J., & Dagless, E. (1994). Petri-net-based algorithms for parallel-controller synthesis. *IEE Proceedings—Computers and Digital Techniques*, *141*(6), 405–412.
7. Bukowiec, A. (2012). Synthesis of FSMs based on architectural decomposition with joined multiple encoding. *International Journal of Electronics and Telecommunications*, *58*(1), 35–41.
8. Doligalski, M. (2012). *Behavioral specification diversification of reconfigurable logic controllers* (Vol. 20)., Lecture notes in control and computer science Zielona Góra: University of Zielona Góra Press.
9. Foster, H., Krolnik, A., & Lacey, D. (2004). *Assertion-based design* (2nd ed.). Norwell: Kluwer Academic Publishers.
10. Gallier, J. H. (1985). *Logic for computer science: foundations of automatic theorem proving*. New York: Harper & Row Publishers.
11. Girault, C., & Valk, R. (2003). *Petri nets for system engineering: a guide to modeling, verification, and applications*. Berlin: Springer.
12. Gniewek, L., & Kluska, J. (2004). Hardware implementation of fuzzy Petri net as a controller. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, *34*(3), 1315–1324.
13. Jensen, K., Kristensen, K., & Wells, L. (2007). Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, *9*(3), 213–254.
14. Karatkevich, A. (2007). *Dynamic analysis of Petri net-based discrete systems* (Vol. 356). Lecture notes in control and information sciences Berlin: Springer.

15. Kozłowski, T., Dagless, E., Saul, J., Adamski, M., & Szajna, J. (1995). Parallel controller synthesis using Petri nets. *IEE Proceedings—Computers and Digital Techniques*, *142*(4), 263–271.
16. Łabiak, G., Adamski, M., Doligalski, M., Tkacz, J., & Bukowiec, A. (2012). UML modelling in rigorous design methodology for discrete controllers. *International Journal of Electronics and Telecommunications*, *58*(1), 27–34.
17. Tkacz, J., & Adamski, M. (2012). Logic design of structured configurable controllers. In *Proceedings of IEEE 3rd international conference on networked embedded systems for every application NESEA'12* (p. 6). Liverpool, United Kingdom

Architectural Synthesis of Petri Nets

Arkadiusz Bukowiec

Abstract New methods of Petri net array-based architectural synthesis are presented. Methods are based on the parallel decomposition of control algorithm into concurrently working state machine subnets and structural decomposition of a digital system. Structural decomposition leads to realization of a logic circuit as a two-level structure, where the combinational circuit of the first level is responsible for firing of transitions, and the second level is a memory used for generation of micro-operations. The memory organization depends on selected architecture. State machine subnets are determined by colors. Places are encoded using minimal numbers of bits. Micro-operations assigned to places are written in memory. Such an approach allows modular organization of logic circuit where each block has strictly determined function and balanced usage of different kinds of resources available in modern FPGAs.

Keywords Digital circuits synthesis · FPGAs · Logic controllers · Petri nets

1 Introduction

Petri nets (PNs) [16, 23] are one of the popular design entries used in formal synthesis and logic synthesis of the application specific logic controllers [4, 12, 13, 15, 19]. Field programmable gate arrays (FPGAs) are very often used for implementation of such control systems. Typically, the Petri net diagrams are translated into behavioral HDL descriptions [2, 14, 25] and then implemented into FPGA devices using one-hot place encoding where each single place is represented by one flip-flop [1]. Such an approach requires hardware implementation of a large number of logic functions and flip-flops included in logic blocks. Furthermore, this approach causes a flat realization of logic circuit with one large block responsible for generation of all functions.

A. Bukowiec (✉)
Institute of Computer Engineering and Electronics, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: a.bukowiec@iie.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_8

Together, it leads to a functional decomposition during the synthesis process and consumes a large number of look-up tables (LUTs).

One of the methods of obtaining the well structured logic circuits is application of architectural decomposition [3, 7, 18, 21]. It leads to a multi-level structure, where each single block is responsible for a specific function. Additionally, it allows usage of logic elements together with embedded memory blocks [6, 24] and in this way it forces balanced utilization of FPGA resources.

This chapter presents the methods of Petri net synthesis. The Petri net is initially colored [15, 27]. Places that are colored by the same color create one state machine (SM) subnet and they are encoded by a minimal-length binary code. It leads to parallel decomposition of a control algorithm, and it also causes parallel decomposition of a logic circuit. The logic functions describing the behavior of the Petri net are grouped into two sets. The first set contains functions responsible for firing of transitions and the second one contains functions responsible for generation of microoperations. Such classification allows application of architectural decomposition where the first set is going to be synthesized in LUTs and implemented as a combinational circuit of the first level, and the second set is going to be realized with the use of the embedded memory blocks as a decoder of the second level.

2 Synthesis Methods

2.1 Main Idea

The idea of the presented synthesis methods is based on parallel decomposition of a Petri net into concurrent SM subnets and architectural decomposition of a logic circuit. Two approaches to the parallel decomposition are described:

- with doublers of macroplaces [9],
- with one common wait place [8].

The architectural decomposition leads to realization of a logic circuit in a two-level structure, where the combinational circuit of the first level is responsible for transition firing, and the second level is implemented as a memory with decoder and is used for generation of the microoperations. Two variants of organization of the decoder are as follows:

- with one shared operational memory for all SM-subnets [8],
- with many flexible distributed memories, one for each SM-subnet [10].

Variants of Petri net parallel decomposition and organization of the decoder can be combined together without any restrictions, which leads to four possible methods of architectural synthesis.

2.2 Architecture

Logic circuit implementing a Petri net (Fig. 1) is build of i two-level parallel chains, where each chain is responsible for one SM-subnet. The chains are obtained as a result of parallel decomposition. $i = 1, 2, \ldots, I$, where I is a number of colors in the Petri net (each color corresponds to one SM-subnet). Places are encoded separately in each SM-subnet, and architectural decomposition is applied to each chain. Operations assigned to the places are placed in memory. The combinational circuits (CC^i) of the first level are responsible for generation of the excitation functions:

$$D^i = D^i(X, Q), \quad (1)$$

where $Q = Q^1 \cup Q^2 \cup \cdots \cup Q^I$ is the set of variables used to store the codes of marked places. The memory of the circuit is build from I D-type registers RG^i which hold a code of current place of each SM subnet. The second level decoder Y is responsible for generation of operations and it is implemented using memory blocks. Its functionality can be described by the function

$$Y = Y(Q) \quad (2)$$

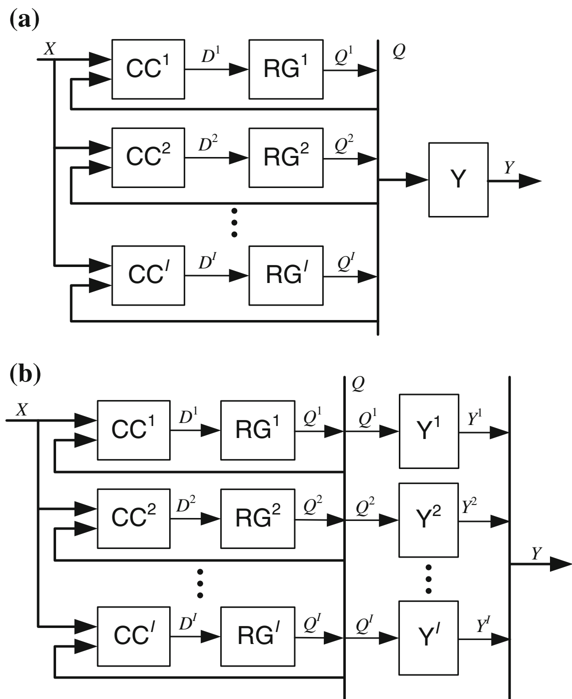


Fig. 1 Logic circuits of a Petri net. **a** With shared operational memory. **b** With flexible distributed memories

in case of one shared operational memory for all SM subnets (Fig. 1a), or by the function

$$Y^i = Y^i(Q^i). \tag{3}$$

in case of many flexible distributed memories (Fig. 1b).

2.3 Synthesis Steps

The entry point to the synthesis method is a colored interpreted Petri net with outputs of Moore type. There are many algorithms of coloring of Petri nets, for example, such as described in [27, 28] or in chapter "Symbolic Coloring of Petri Nets" of this book. The whole synthesis process includes following steps:

1. *Formation of subnets.* The purpose of this step is to extract SM-subnets from the Petri net by application of parallel decomposition. Let us assume that the Petri net is colored with I different colors. Let us start the decomposition from the first color ($i = 1$). All places colored by this color create the first SM-subnet. This SM-subnet does not contain a wait place or any doublers. So, it is the same for both approaches of decomposition. Next subnets are created in a similar way. But, places which have been previously selected by already created SM-subnets have to be replaced by doublers of macropalces or a wait place, depending on parallel decomposition approach. First, in both approaches, all sequences of such places are replaced by macroplaces. Then, in the first approach, the doublers of these macroplaces appear in a new SM-subnet and they do not have any output signals assigned. There can be several doublers of macroplaces in one subnet but if some of them occur in a sequence then they can be replaced by one doubler. In the second approach, the macroplaces are removed and replaced by one wait place. It creates a more complicated net but with less number of places. It could be a benefit during the encoding process. The examples of application of both parallel decompositions are shown in Fig. 2. Figure 2a shows an initially colored Petri net. It is colored by two colors: C_1 and C_2. The first SM-subnet (Fig. 2b) is the same for both approaches of parallel decomposition and consists of all places colored by color C_1. The second SM-subnet includes two doublers of macroplaces DP_1 and DP_2 (Fig. 2c) if the first approach is applied, or one wait place WP_0 (Fig. 2d) if the second one is applied. The doublers or wait place are treated as normal places in the next steps. It should be also mentioned that there are other algorithms for SM-subnets extraction, like described in [17, 26, 29], but they have to be adapted to this synthesis method in case of usage at this step.
2. *Encoding of places.* The purpose of this step is to assign a binary code to each place. The encoding is done using minimal number of required bits. One-hot encoding [22] is not acceptable in this method because the place code is also an address of operation memory. Places are encoded separately in each SM-subnet, obtained in previous step. It is required to use

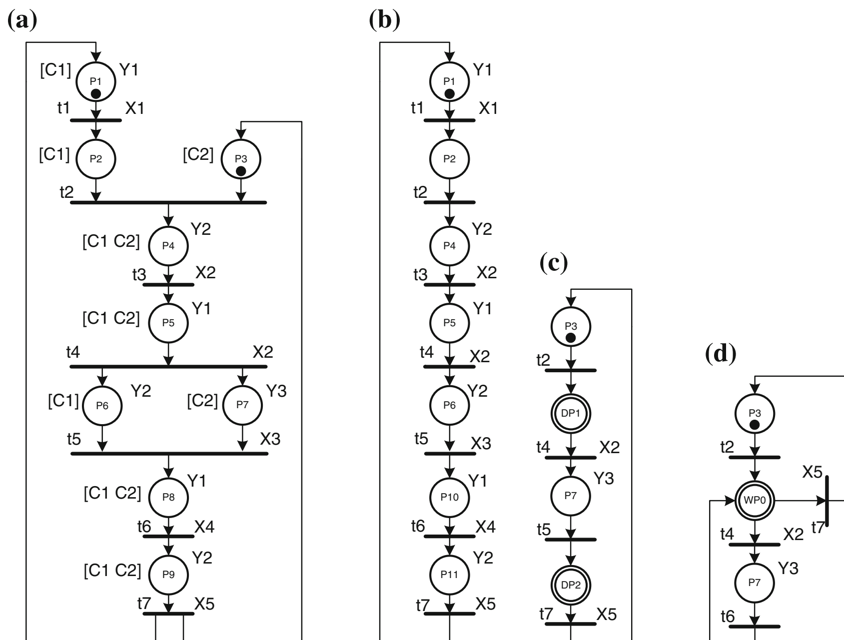


Fig. 2 Application of parallel decompositions

$$R_i = \lceil \log_2 |P_i| \rceil \tag{4}$$

bits to encode them, where $P_i \subseteq P \cup MP_i$ is a set of places in a subnet that was created based on the color C_i. The set MP_i consists of doublers of macroplaces or wait place added to this subnet. The variables from set $Q^i \subset Q$ are used to store this code, where $Q = \{q_0, \ldots, q_{R-1}\}$,

$$R = \sum_{i=1}^{I} R_i \tag{5}$$

and $Q^i = \{q_{\rho - R_i}, \ldots, q_{\rho-1}\}$, where $\rho = \sum_{\iota=1}^{i} R_\iota$. Places that belong to the initial marking set M_0 get code equal to 0. If the subnet does not include any place from the initial marking set M_0 the code equal to 0 should be assigned to the wait place or to the doubler of macroplace that has replaced such place.

3. *Formation of conjunctions.* Conjunctions describe places, transitions and the place conditions. They are needed for easier creation of equations that describe the systems (1). A conjunction describing place p consists of affirmation or negation of variables q_r that are used to store the code of this place. If the code has 0 in the rth bit then negation is used, and if it has 1 then affirmation is used. A conjunction describing transition t consists of place conjunctions of input places

to this transition and a condition φ assigned to this transition. The hold of place p condition conjunction consists of negation of sum of transition conjunctions of all its output transitions and its place conjunction.

4. *Formation of logic equations*. Logic equations describe functions (1) of combinational circuits CC^i. They are created according to D flip-flop equation and they are build of transition conjunctions and hold of place condition conjunctions. If the variable q_r is set to 1 in the code of place p then the sum of corresponding variables D_r consists of transition conjunctions of all input transitions of p and the hold of the place condition conjunctions.
5. *Formation of memory content*. The memory content can be described as the equations or as a table. It is required to form one table in case of one shared operational memory or I tables in case of flexible distributed memories, respectively, according to the systems (2) or (3). Tables always consist of two columns. First column is an address and it is described by variables $q_r \in Q$ or $q_r \in Q^i$. In case of one shared operational memory these variables represent superposition of codes of all states from all subnets, not only allowed markings. The second column is an operation. The operation is represented by output variables form the set Y or form the set $Y^i \subseteq Y$, where the set Y^i consists only of variables y_n that are under control of the SM-subnet formed by the color C_i. There should be only the variables that are in the elementary conjunctions ψ associated to the places described by current address. In case of one shared operational memory this table very often is very long because it has 2^R lines. The other way to describe the memory content is to create a set of logic equations to describe output variables. They describe output variable as a sum of place conjunctions of places corresponding to the elementary conjunctions ψ that consist of corresponding output variables. Such equations can be formed in both cases of organization of memory. However, it is recommended to form equations for one shared operational memory and to form tables for flexible distributed memories.
6. *Formation of logic circuit and implementation*. This step describes the rules of creation of an HDL description of Petri net model and its implementation in an FPGA device. A bottom-up approach is applied. First, there should be created separate modules/entities for each CC^I, RG^I, and Y^I or Y block. Places and transitions conjunctions can be described using standard bit-wise operators. Then logic equations can be described with use of these conjunctions also using bit-wise operators. The module/entity for each CC^i block should have X and Q inputs and D^i outputs. The register RG^i should be described as R_i-bits D-type register with asynchronous reset. The typical synthesis template can be used [20]. The memory Y can be described as the logic equations. Memories Y^I can be described as the processes with the `case` statement. To synthesize memories with utilization of embedded memory blocks it is required to add special synthesis directive. The syntax of this directive depends on FPGA vendor. As far as typically the embedded memory blocks are synchronous, it is also required to create a clock input and a synchronous reset. The top-level module should describe connections of all modules according to the logic schematics presented in Fig. 1. Additionally the global reset signal should be connected to reset inputs of registers and to the reset

of memory. The global clock signal should be connected to the registers and the memory. The memory should be trigged by an opposite edge than the registers. It allows operations to be generated in one clock cycle [6]. Such model of logic circuit can be passed to the third-party synthesis and implementation tools.

3 Implementation of Synthesis Method

The presented algorithms were implemented in C# in Microsoft .NET environment as a standalone library called *Decompone And VHDL Code Gen* [11]. The whole process of synthesis is fully automated and does not require any interaction with user. The user is only obligated to choose the method of parallel decomposition into SM-components with doublers of macroplaces or with one wait place and to choose the organization of decoder with one shared operational memory or flexible distributed memories in the beginning of the synthesis process. The whole algorithm was implemented with use of three classes (Fig. 3). The main class is *GenerateHDL*. It includes public method *void VHDLCODEGenerate()* to run the whole process and to generate the VHDL code. The entry point is an object-oriented model of colored Petri net [11] passed to the constructor *GenerateHDL(PetriNet net, Sting netName, String options)* of this class. The other two parameters of *String* type are the name of the Petri net and options that describe the methods of parallel decomposition and

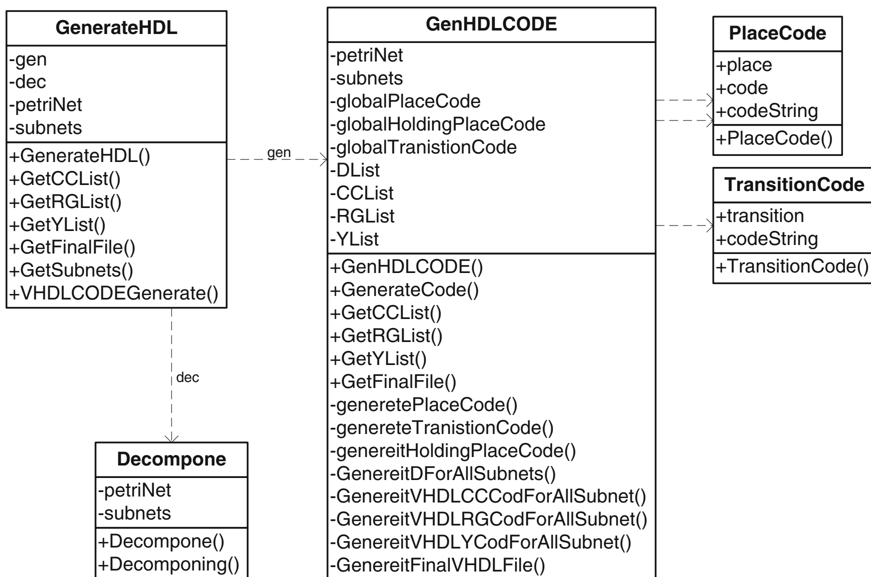


Fig. 3 Class diagram of *Decompone And VHDL Code Gen* library

of organization of the decoder. The generated VHDL entities are obtained by the execution of the *String GetFinalFile()* method. There are also other public methods *List⟨String⟩ GetCCList()*, *List⟨String⟩ GetRGList()*, and *List⟨String⟩ GetYList()* to obtain separate files for each entity, and *List⟨PetriNet⟩ GetSubnets()* to obtain the SM-subnets as the object-oriented models. The last method can be used for further processing of the SM-subnets without the synthesis. Other methods, which perform particular steps of the synthesis algorithm, are internal and not available for an end-user. They are invoked automatically by the main method.

This synthesis process is divided into two parts: parallel decomposition of Petri net (1st step)—implemented in the *Decompone* class, and creation of VHDL description (6th step)—implemented in the *GenHDLCODE* class. The second part includes encoding of places (2nd step) and formation of conjunctions (3rd step), logic equations (4th step), and memory contents steps (5th step).

The main synthesis process starts from the parallel decomposition of Petri net into SM-subnets. It is done by the *List⟨PetriNet⟩ Decomponing()* method from the *Decompone* class invoked on *Decompone dec* property. The Petri net has to be passed into constructor *Decompone(PetriNet net)* of this property.

The other steps of the synthesis algorithm are performed by the methods from the *GenHDLCODE* class. All equations are generated directly into VHDL syntax. It allows an easy and fast generation of description of the model of logic circuit in VHDL. To start these steps the *Boolean GenerateCode()* method on the *GenHDLCODE gen* property has to be invoked. The Petri net and SM-subnets are passed into constructor *GenHDLCODE(PetriNet net, List⟨PetriNet⟩ subnets)* of this property. The *GenerateCode* method runs all steps in sequence: *void generetePlaceCode(), void genereteTranistionCode(), void genereitHoldingPlaceCode(), void GenereitDForAllSubnets(), void GenereitVHDLCCCodForAllSubnet(), void GenereitVHDLRGCodForAllSubnet(), void GenereitVHDLYCodForAllSubnet(), void GenereitFinalVHDLFile()*. These methods run the following steps of synthesis algorithm and store the results in the private lists. Codes, conjunctions and equations are generated by first four methods. VHDL files are build up from these expressions. The *GenereitVHDLCCCodForAllSubnet* method generates the files describing combinational circuits and creates one VHDL file for each subnet. The *GenereitVHDLRGCodForAllSubnet* method creates one register for each subnet. The registers only differ in the length of vector and they are generated based on the synthesis template [5]. The *GenereitVHDLYCodForAllSubnet* method generates all decoders in VHDL. Finally, the *GenereitFinalVHDLFile* method creates the top-level module based on structure of logic circuit (Fig. 1). Such created model of logic circuit can be passed to the third-party synthesis and implementation tools.

4 Conclusion

A method of architectural synthesis and implementation of application specific logic controllers into FPGAs was presented in this chapter. A logic circuit of ASCL is obtained by application of parallel and structural decompositions. Two variants of

each decomposition have been presented. Next, a special method of logic synthesis is proposed. The digital design is based on minimal encoding of places in each SM-subnet. Additionally, output functions are extracted to autonomic blocks and they can be implemented with the use of embedded memory blocks. It leads to reasonable usage of different kinds of logic resources of FPGA devices. This method of synthesis is dedicated to highly concurrent Petri nets with a large number of inputs and outputs. It provides special benefits when output equations are complicated comparing with application of the standard method of synthesis.

The presented method was implemented and compiled into the standalone library that can be used in other CAD systems. Now, the colored Petri net as a graphic representation of algorithm is used as an entry point to the synthesis method. The usage of designed library is fully automated, so it could be easily integrated with design tools in any CAD system. As the output a set of the VHDL files is generated. These files describe a logic circuit of ASCL and can be used in any commercial CAD system or synthesis and implementation tools.

References

1. Adamski, M., & Węgrzyn, M. (2009). Petri nets mapping into reconfigurable logic controllers. *Electronics and Telecommunications Quarterly*, *55*(2), 157–182.
2. Adamski, M., Węgrzyn, M., & Wolański, P. (1998). A VHDL based Approach to Logic Controllers Design. In *Proceedings of International Conference Programmable Devices and Systems PDS'98* (pp. 9–16). Gliwice, Poland.
3. Barkalov, A., Titarenko, L., Malcheva, R., & Soldatov, K. (2013). Hardware reduction in FPGA-based Moore FSM. *Journal of Circuits, Systems and Computers*, *22*(3), 1350006.
4. Biliński, K., Adamski, M., Saul, J., & Dagless, E. (1994). Petri-net-based algorithms for parallel-controller synthesis. *IEE Proceedings – Computers and Digital Techniques*, *141*(6), 405–412.
5. Brown, S., & Vernesic, Z. (2005). *Fundamentals of digital logic with VHDL design* (2nd ed.). New York: McGraw-Hill.
6. Bukowiec, A. (2009). *Synthesis of finite state machines for FPGA devices based on architectural decomposition* (Vol. 13). Lecture notes in control and computer science. Zielona Góra: University of Zielona Góra Press.
7. Bukowiec, A. (2012). Synthesis of FSMs based on architectural decomposition with joined multiple encoding. *International Journal of Electronics and Telecommunications*, *58*(1), 35–41.
8. Bukowiec, A., & Adamski, M. (2012). Logic synthesis for FPGAs of interpreted Petri net with common operation memory. In Z. Bradáč, F. Bradáč, & F. Zezulka (Eds.), *11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems PDeS 2012* (pp. 57–62). IFAC-PapersOnLine. Brno, Czech Republic.
9. Bukowiec, A., & Adamski, M. (2012). Synthesis of macro Petri nets into FPGA with distributed memories. *International Journal of Electronics and Telecommunications*, *58*(4), 403–410.
10. Bukowiec, A., & Adamski, M. (2012). Synthesis of Petri nets into FPGA with operation flexible memories. In *Proceedings of the IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS'12* (pp. 16–21). Tallinn, Estonia.
11. Bukowiec, A., Tkacz, J., Gratkowski, T., & Gidlewicz, T. (2013). Implementation of algorithm of Petri nets distributed synthesis into FPGA. *International Journal of Electronics and Telecommunications*, *59*(4), 317–324.

12. Cortés, L. A., Eles, P., & Peng, Z. (2003). Modeling and formal verification of embedded systems based on a Petri net representation. *Journal of Systems Architecture*, *49*(12–15), 571–598.
13. Gniewek, L., & Kluska, J. (2004). Hardware implementation of fuzzy Petri net as a controller. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, *34*(3), 1315–1324.
14. Gomes, L., Costa, A., Barros, J., & Lima, P. (2007). From Petri net models to VHDL implementation of digital controllers. In *33rd Annual Conference of the IEEE Industrial Electronics Society IECON'07* (pp. 94–99). Taipei, Taiwan: IEEE.
15. Jensen, K., Kristensen, K., & Wells, L. (2007). Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, *9*(3), 213–254.
16. Karatkevich, A. (2007). *Dynamic analysis of petri net-based discrete systems* (Vol. 356). Lecture notes in control and information sciences. Berlin: Springer.
17. Karatkevich, A., & Wiśniewski, R. (2012). Computation of Petri nets covering by SM-components based on the graph theory. *Przeglad Elektrotechniczny*, *88*(8), 141–144.
18. Khamis, A., Zydek, D., Borowik, G., & Naidu, D. S. (2013). Control system design based on modern embedded systems. In R. Moreno-Díaz, F. R. Pichler, & A. Quesada-Arencibia (Eds.), *Computer aided systems theory - EUROCAST 2013* (Vol. 8112, pp. 491–498). Lecture notes in computer science. Berlin: Springer.
19. Latorre-Biel, J.-I., Jiménez-Macías, E., Pérez de la Parte, M., Blanco-Fernández, J., & Martínez-Cámara, E. (2014). Control of discrete event systems by means of discrete optimization and disjunctive colored PNs: Application to manufacturing facilities. *Abstract and Applied Analysis*, *2014*, 821707.
20. Lee, J. M. (1999). *Verilog QuickStart: A practical guide to simulation and synthesis in Verilog*. Norwell: Kluwer Academic Publishers.
21. Łuba, T., Borowik, G., & Kraśniewski, A. (2009). Synthesis of finite state machines for implementation with programmable structures. *Electronics and Telecommunications Quarterly*, *55*(2), 183–200.
22. Marranghello, N., Mirkowski, J., & Bilinski, K. (2000). Synthesis of synchronous digital systems specified by Petri nets. In A. Yakovlev, L. Gomes, & L. Lavagno (Eds.), *Hardware design and Petri nets* (pp. 129–150). Boston: Kluwer Academic Publishers.
23. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.
24. Rawski, M., Borowik, G., Łuba, T., Tomaszewski, P., & Falkowski, B. (2010). Logic synthesis strategy for FPGAs with embedded memory blocks. *Przeglad Elektrotechniczny*, *86*(11a), 94–101.
25. Soto, E., & Pereira, M. (2005). Implementing a Petri net specification in a FPGA using VHDL. In M. Adamski, A. Karatkevich, & M. Węgrzyn (Eds.), *Design of embedded control systems* (pp. 167–174). New York: Springer.
26. Stefanowicz, Ł., Adamski, M., & Wiśniewski, R. (2013). Application of an exact transversal hypergraph in selection of SM-components. In L. Camarinha-Matos, S. Tomic, & P. Graça (Eds.), *Technological innovation for the internet of things* (Vol. 394, pp. 250–257). IFIP advances in information and communication technology. Berlin: Springer.
27. Tkacz, J. (2007). State machine type colouring of Petri net by means of using a symbolic deduction method. *Measurement Automation and Monitoring*, *53*(5), 120–122.
28. Węgrzyn, A. (2006). On decomposition of Petri net by means of coloring. In *Proceedings of IEEE East-West Design & Test Workshop EWDTW'06* (pp. 407–413). Sochi, Russia.
29. Wiśniewski, R., Stefanowicz, Ł., Bukowiec, A., & Lipiński, J. (2014). Theoretical aspects of Petri nets decomposition based on invariants and hypergraphs. In J. J. Park, S.-C. Chen, J.-M. Gil, & N. Y. Yen (Eds.), *Multimedia and ubiquitous engineering* (Vol. 308, pp. 371–376). Lecture notes in electrical engineering. Berlin: Springer.

Decomposition-Based Methods for FSM Implementation

Mariusz Rawski, Piotr Szotkowski and Paweł Tomaszewicz

Abstract Designing a complex digital system requires an effective method for modeling the sequential part of the system. One of the methods is the Finite State Machine based modeling. The implementation efficiency of the sequential part of the designed system has usually a great impact on the processing performance of the whole digital system. Petri nets, which are another method of modeling the sequential part of systems, can also be transformed into FSM-based models. Thus, development of effective synthesis methods for FSM implementation is very important. Digital systems are often implemented in FPGA architectures. Because of their specific structure, the most efficient synthesis methods are based on functional decomposition. This chapter discusses decomposition-based methods for FSM implementation targeting programmable structures.

Keywords FSM · Symbolic function decomposition · Logic synthesis

1 Introduction

Finite state machines (FSM) are an important element used in digital system design [1, 6, 15, 24]. A typical process of FSM implementation includes restructuring methods, such as minimization of internal states, which are independent of the target architecture type in which the FSM is to be implemented. Following that, there is the state encoding stage, which enables the generation of a logic description of the state machine. The state machine can then be realized, as efficiently as possible,

M. Rawski (✉) · P. Szotkowski · P. Tomaszewicz
Institute of Telecommunications, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland
e-mail: rawski@tele.pw.edu.pl

P. Szotkowski
e-mail: p.szotkowski@tele.pw.edu.pl

P. Tomaszewicz
e-mail: p.tomaszewicz@tele.pw.edu.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_9

in the resources of the target architecture using various methods of logic synthesis. The quality of the synthesis method used at this stage affects, to a large extent, the quality of the FSM's hardware implementation. This effect is especially noticeable in the case of implementations targeting Field-Programmable Gate Array (FPGA) devices. The reason for this is the imperfection of currently used technology mapping methods. Many of them are algorithms adapted from the synthesis of logic circuits, designed for standard cell technology—such as logic minimization or factorization of Boolean functions. Such methods transform Boolean expressions represented as a sum of products (SOP) into a multi-level, highly factorized structure, which is, then, mapped into logic cells of an FPGA architecture. Such an approach is incompatible with the nature of the look-up table (LUT) based logic elements of FPGAs, which—from the point of view of logic synthesis—are able to realize any function of a limited number of input variables. Therefore, for programmable structures functional decomposition is a much more efficient method of logic synthesis. The effectiveness of this method for the synthesis of combinational circuits for programmable devices has been repeatedly confirmed in many papers [8, 9, 26, 27].

The application of the concept of decomposition for the synthesis of FSMs was discussed in the literature as early as in the middle of last century. In [15] an *Algebraic Structure Theory of Sequential Machines* has been proposed to solve problems of encoding and decomposition of the FSM. The decomposition was based on division of the original machine into smaller state machines connected in serial or parallel and cooperating with each other to obtain the behavior of the original FSM. This type of decomposition can be called structural decomposition. The computational complexity of this method is so large, that it makes it impossible to apply it to automata with higher number of states. Structural decomposition concept was studied in [3–5, 14].

In the synthesis of finite sate machines, the most important decision to make is the binary encoding of the symbolic states of the FSM. After this stage, the automaton is modeled as a combinational next-state and output function and a state register. Bad state code assignment leads to formation of an FSM that cannot be efficiently implemented by using advanced logic synthesis methods, which usually means that the hardware implementation would require too much logic resources, will be too slow and often will have both of these drawbacks. Most research and publications devoted to the search for *optimal* state machine encoding methods assume the use of the minimum number of internal states, and sometimes the minimum number of encoding bits—which leads to the minimum number of flip-flops in the hardware implementation. Sometimes a better solution turns out to be to *sacrifice* some additional internal states or flip-flops in order to get a faster operating hardware implementation. Unfortunately, the only way to obtain an optimal state encoding of the finite state machine is to check all possible solutions [5], and since it is a NP-complete problem, this type of approach requires computation time unacceptable even for small FSMs.

Most state encoding methods are not adapted to modern programmable structures, such as FPGAs. Traditional methods for constructing the next-state and output function of the FSM introduce an assignment of code words to FSM's states referred to in literature as highly-encoded states. Finite state machines with such an encoding

usually require a minimum number of flip-flops to implement the state register and the next state functions; the computation of each state encoding bit depends on most of the bits of the current state and input variables—such functions are called wide combinational functions. Such features of an encoded state machine are acceptable in implementations in standard cells technology or Programmable Logic Array (PLA) structures. However—because the FPGA structures are usually composed of a large number of flip-flops and their logic cells can realize functions of a small number of variables (narrow combinational functions)—the application of this type of encoding can lead to a very inefficient implementation, both in terms of required resources and operating speed. The only method leading to highly-encoded states, which has been developed in detail and could be used for FPGA structures, is the method proposed in [15]. However, because of very high computational complexity, it has never been implemented.

First attempts to develop state assignment algorithms feasible for an implementation in the form of computer applications date back to 1960. They imitated the methods used by designers, formulated by Humphrey in [16] in the form of rules of state code adjacency. Applying these principles in [2, 12] yielded algorithms that are designed to minimize the number of products of SOP expressions describing the combinational next-state and output function of the FSM. Although technologies of digital circuit implementation have changed over the years, the rules formulated by Humphrey became the basis for many generations of state code assignment algorithms.

A novel approach based on symbolic minimization has been used in *Kiss* [10] and its successor *Nova* [36], both designed for two-level implementations, suitable e.g., for PLA technology. The main idea applied in these algorithms was to perform logic minimization before the states encoding phase. At the stage of logic minimization authors of the method benefit from the fact that for two-level implementations the function of the implementation cost can be easily defined. It can be estimated based on the number of inputs, outputs and the number of products in a minimized form of the Boolean function.

Unfortunately, for a multi-level implementation it is impossible to propose such a simple estimation of the implementation cost. In the case of multi-level realizations a whole group of new challenges appears. The greatest potential of these implementations—i.e., the ability to apply different trade-offs between the amount of resources required for implementation, operation speed, power consumption, etc.—introduces completely new, very complex criteria for automatic synthesis algorithms. This greatly complicates the process of hardware implementation of FSMs and makes it very difficult to define a function that assesses the quality (cost) of implementation.

Most encoding methods proposed for multi-level implementations are designed to generate internal states encoding yielding next-state and output function *easy* for mapping by the tool performing the multi-level combinational logic synthesis. *Mustang* is one of the earliest methods of this kind [11]. It has been designed to work with the logic synthesis system *MIS* [7]. The following two other methods are examples of this approach: *Jedi* [21] and *Muse* [13]. Another method—one-hot encoding—creates a state machine with one flip-flop for each state, which reduces

the width of the next-state and output functions (at the cost of the number of bits used for encoding). Such representation makes the FSM possible to implement effectively in FPGAs, thus this method is usually preferred for the realization of complex finite state machines in programmable architectures.

All these methods assume a strong correlation between the size of SOP expressions (or their factorized forms) describing the Boolean function and the quality of the hardware implementation of the function. However, the heuristics used in conventional synthetic tools are not effective in the case of modern technologies, such as programmable FPGA structures. This is due to the structure of the logic cell, which is the basic building block in this technology.

As has been already mentioned, the functional decomposition is considered to be the most effective method of logic synthesis for FPGA structures. However—for this type of multi-level synthesis—it is extremely difficult to specify what *an easy to implement function* means. Therefore, there is a high probability that the proposed states encoding will not be compatible with the later stage of the next-state and output function synthesis based on the functional decomposition.

In [19] there has been proposed a method of states coding *Secod*, which uses optimization criteria based on the concept of information measures. This method is intended to work with the logic synthesis algorithm based on functional decomposition, which also uses the information measures. Therefore, both methods are consistent mechanisms for implementation of sequential circuits in an FPGA.

All the previously discussed methods of hardware implementations of FSMs have a common feature; their implementation process is divided into two stages: states encoding and synthesis of the encoded next-state and output function (to map it in the resources of the target technology). In [25] a novel approach to FSM synthesis for FPGA structures has been proposed—**a symbolic functional decomposition**. This concept eliminates the two-stage division of finite state machine synthesis. The proposed method operates on a symbolic next-state and output function of the FSM, introducing partial encoding of the state variable for the subsequent stages of the multi-level synthesis, so as to obtain the best decomposition at a given stage. With this approach the state encoding is introduced gradually at every consecutive decomposition iteration of mapping of the FSM into FPGA logic cells.

2 Basic Information

2.1 Blanket Calculus in FSM Modeling

Definition 1 (*Finite State Machine*) A finite state machine A is an algebraic system defined by $A = \langle I, O, S, \delta, \lambda \rangle$, where I is the set of input symbols, O is the set of output symbols, S is the set of internal states, $\delta : S \times I \to S$ describes the next-state function and $\lambda : S \times I \to O$ describes the output function.

Usually input and output symbols are encoded by binary input variables $X = \{x_1, \ldots, x_n\}$ and binary output variables $Y = \{y_1, \ldots, y_m\}$, while the internal states are encoded by a symbolic variable, which acts as the input (current state) Q and the output (next state) Q'. Typically, an FSM is represented as a state transition table, where each row consists of an input cube (representing the value of the input variables $x_1, \ldots, x_n$), a current state, a next state and an output cube (representing the value of the output variables $y_1, \ldots, y_n$). Table 1a presents the state transition table of an example FSM.

Blanket algebra can be used to describe logic dependencies in such an FSM. The below reviews only some information concerning blanket algebra; a more detailed description of blanket calculus can be found in [8].

Definition 2 (*Blanket*) A blanket on a set S is a collection $\beta = \{B_1, \ldots, B_k\}$ of nonempty and distinct subsets of S, called blocks, whose union is S.

Define nonempty operator ne so that for any collection S_i of subsets of S, $ne\{S_i\}$ is S_i with empty blocks removed (if any were present in S_i) and with only one copy of each block, if the set contained many copies of the same block.

Table 1 (a) State transition table of an example FSM, (b) binary encoding of the Q state variable, (c) symbolic encoding of the Q state variable

(a)								(b)			(c)	
	x_1	x_2	x_3	x_4	Q	Q'	y_1	q_1	q_2	q_3	Q_U	Q_V
1	1	0	0	0	s_1	s_1	0	0	0	0	u_1	v_1
2	0	1	0	0	s_1	s_1	0	0	0	0	u_1	v_1
3	0	0	1	0	s_1	s_2	0	0	0	0	u_1	v_1
4	0	0	0	1	s_1	s_2	0	0	0	0	u_1	v_1
5	1	0	0	0	s_2	s_2	1	0	0	1	u_2	v_2
6	0	1	0	0	s_2	s_3	1	0	0	1	u_2	v_2
7	0	0	1	0	s_2	s_2	1	0	0	1	u_2	v_2
8	0	0	0	1	s_2	s_1	1	0	0	1	u_2	v_2
9	1	0	0	0	s_3	s_3	1	0	1	0	u_2	v_1
10	0	1	0	0	s_3	s_5	1	0	1	0	u_2	v_1
11	0	0	1	0	s_3	s_3	1	0	1	0	u_2	v_1
12	0	0	0	1	s_3	s_5	1	0	1	0	u_2	v_1
13	1	0	0	0	s_4	s_4	0	0	1	1	u_1	v_3
14	0	1	0	0	s_4	s_2	0	0	1	1	u_1	v_3
15	0	0	1	0	s_4	s_3	0	0	1	1	u_1	v_3
16	0	0	0	1	s_4	s_3	0	0	1	1	u_1	v_3
17	1	0	0	0	s_5	s_5	1	1	0	0	u_2	v_4
18	0	1	0	0	s_5	s_5	1	1	0	0	u_2	v_4
19	0	0	1	0	s_5	s_1	1	1	0	0	u_2	v_4
20	0	0	0	1	s_5	s_4	1	1	0	0	u_2	v_4

Definition 3 (*Product of blankets*) The product $\beta_1 \cdot \beta_2$ of two blankets is:

$$\beta_1 \cdot \beta_2 = ne\{B_i \cap B_j | B_i \in \beta_1 i B_j \in \beta_2\}. \tag{1}$$

For two blankets we write $\beta_1 \leq \beta_2$ (β_1 is smaller than or equal to β_2) if and only if for every block B_i of β_1 there exists a block B_j of β_2 such that $B_i \subseteq B_j$. The $\leq$ relation is reflexive and transitive.

Let $\beta_{x_1}, \ldots, \beta_{x_n}$ be two-block blankets induced by input variables $x_1, \ldots, x_n$, and $\beta_{y_1}, \ldots, \beta_{y_m}$ be two-block blankets induced by output variables $y_1, \ldots, y_m$; let β_Q be a multi-block blanket induced by the current state variable, and $\beta_{Q'}$ be a multi-block blanket induced by the next state variable. The relationship that must hold for the FSM is as follows:

$$\beta_X \cdot \beta_Q \leq \beta_Q \cdot \beta_Y \tag{2}$$

where: $\beta_X = \beta_{x_1} \cdot \beta_{x_2} \cdot \cdots \cdot \beta_{x_n}$ and $\beta_Y = \beta_{y_1} \cdot \beta_{y_2} \cdot \cdots \cdot \beta_{y_m}$.

Example 1 Application of blanket calculus for description of logic dependencies in the FSM from Table 1a.

$$\begin{aligned}
\beta_X &= \beta_{x1} \cdot \beta_{x2} \cdot \beta_{x3} \cdot \beta_{x4} \\
&= \{\overline{1, 5, 9, 13, 17}; \overline{2, 6, 10, 14, 18}; \overline{3, 7, 11, 15, 19}; \overline{4, 8, 12, 16, 20}\},
\end{aligned} \tag{3}$$

$$\begin{aligned}
\beta_Y &= \beta_{y1} \\
&= \{\overline{1, 2, 3, 4, 13, 14, 15, 16}; \overline{5, 6, 7, 8, 9, 10, 11, 12, 17, 18, 19, 20}\},
\end{aligned} \tag{4}$$

$$\beta_Q = \{\overline{1, 2, 3, 4}; \overline{5, 6, 7, 8}; \overline{9, 10, 11, 12}; \overline{13, 14, 15, 16}; \overline{17, 18, 19, 20}\}, \tag{5}$$

$$\beta_{Q'} = \{\overline{1, 2, 8, 19}; \overline{3, 4, 5, 7, 14}; \overline{6, 9, 11, 15, 16}; \overline{13, 20}; \overline{10, 12, 17, 18}\}, \tag{6}$$

$$\beta_X \cdot \beta_Q = \{\overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6}; \overline{7}; \overline{8}; \overline{9}; \overline{10}; \overline{11}; \overline{12}; \overline{13}; \overline{14}; \overline{15}; \overline{16}; \overline{17}; \overline{18}; \overline{19}; \overline{20}\}, \tag{7}$$

$$\beta_{Q'} \cdot \beta_Y = \{\overline{1, 2}; \overline{8, 19}; \overline{3, 4, 14}; \overline{5, 7}; \overline{6, 9, 11}; \overline{15, 16}; \overline{13}; \overline{20}; \overline{10, 12, 17, 18}\}. \tag{8}$$

It can be easily shown that (7) and (8) satisfy condition (2).

2.2 FSM State Encoding

Hardware implementations use binary logic, so in order to implement an FSM in a digital system the Q symbolic variable must be encoded. Encoding of the symbolic variable means replacing it by a set of binary variables in a way that reproduces the symbolic description in the physical system. Hence the state encoding is the process of assigning a unique combination of binary variables to each value s from the set S of the Q symbolic variable. To generate a sufficient number of codes, $k \geq \lceil \log_2 |Q| \rceil$ binary variables have to be used, where $|Q|$ is the cardinality of the Q set. Thus, the state encoding is a mapping of values S of the Q symbolic variable onto values of the set of encoding binary variables $q_1, \ldots, q_k$. Since the Q and Q' variables are defined

on the same set of values S, the mapping introduced for the Q variable defines also the mapping of the Q' variable, which is replaced by a set of variables $q'_1, \ldots, q'_k$.

Let $\beta_{q_1}, \ldots, \beta_{q_k}$ be two-block blankets induced by variables $q_1, \ldots, q_k$, and $\beta_{q'_1}, \ldots, \beta_{q'_k}$ be two-block blankets induced by variables $q'_1, \ldots, q'_k$. The FSM state encoding is valid if (9) is satisfied.

$$\beta_{q_1} \cdot \beta_{q_2} \cdot \cdots \cdot \beta_{q_k} = \beta_Q. \tag{9}$$

If (9) is satisfied (10) is also satisfied.

$$\beta_{q'_1} \cdot \beta_{q'_2} \cdot \cdots \cdot \beta_{q'_k} = \beta_{Q'}. \tag{10}$$

In other words, the encoding of internal states can be described as the introduction of such two-block blankets $\beta_{q_1}, \ldots, \beta_{q_k}$, that satisfy the condition (9). Each β_{q_i} blanket can be obtained by merging the appropriate blocks of the β_Q blanket, so as to obtain two blocks. An example of an encoding of the Q state variable for the finite state machine described in Table 1a using three binary variables q_1, q_2, q_3 is shown in Table 1b.

To encode the Q symbolic variable other symbolic variables $Q_1, \ldots, Q_V$ can also be used (symbolic coding). In this case, the $\beta_{Q_1}, \ldots, \beta_{Q_V}$ blankets induced by the symbolic coding variables are multi-block blankets. Their number of blocks depends on the number of symbolic values held by the coding variables. The $\beta_{Q_1}, \ldots, \beta_{Q_V}$ blankets must also satisfy the condition (9) to make the encoding valid.

Example 2 Symbolic encoding of the Q variable for the FSM from Table 1a.

Let Q_U and Q_V be two symbolic variables holding respectively $\{u_1, u_2\}$ and $\{v_1, v_2, v_3, v_4\}$ values (Table 1c). These variables induce the following β_{Q_U} and β_{Q_V} blankets:

$$\beta_{Q_U} = \{\overline{1, 2, 3, 4, 13, 14, 15, 16}^{u_1}; \overline{5, 6, 7, 8, 9, 10, 11, 12, 17, 18, 19, 20}^{u_2}\}, \tag{11}$$

$$\beta_{Q_V} = \{\overline{1, 2, 3, 4, 9, 10, 11, 12}^{v_1}; \overline{5, 6, 7, 8}^{v_2}; \overline{13, 14, 15, 16}^{v_3}; \overline{17, 18, 19, 20}^{v_4}\}. \tag{12}$$

3 Symbolic Functional Decomposition

As it has been already mentioned, the typical method of FSM synthesis consists of two stages: the internal state encoding to obtain a Boolean next-state and output function, and the logic synthesis stage responsible for implementing this function in the most efficient way in the resources of the target architecture. All encoding methods for the multi-level implementation are designed to make the next-state and output function *easy* to implement for the chosen synthesis method. Unfortunately, for functional decomposition based synthesis methods there is no reasonable estimate of the *ease* feature of the encoded next-state and output function.

Symbolic functional decomposition eliminates the division of the FSM synthesis process into the separate stages of state encoding and logic synthesis. The proposed method accepts an FSM description with symbolic states and performs symbolic decomposition maintaining the multi-value representation of the state and next-state variables, effectively encoding them partially on every decomposition step, but only to the extent that is required—and optimal—for the given step.

A symbolic functional decomposition of a finite state machine can be described in a manner similar to that used for modeling serial functional decomposition of combinational functions [8].

Let X be the set of primary input variables, Y be the set of primary output variables of a certain FSM specified by a state transition table. Let Q and Q' be the multi-valued variables representing current and next state of this FSM. Let U and V be two subsets of X such that $U \cup V = X$. Let Q_V and Q_U be the multi-valued variables encoding the Q variable. Let β_V and β_U be blankets induced, respectively, by the primary input subsets V and U, and β_{Q_V}, β_{Q_U} be blankets induced by the multi-valued variables Q_V and Q_U. Let β_Y and $\beta_{Q'}$ be blankets induced by the primary outputs set and by the next state multi-valued variable Q'.

Theorem 1 (Existence of the symbolic functional decomposition) *The FSM has a symbolic functional decomposition with respect to* (U, Q_U, Q_V, V) *iff there exists a blanket* β_G *such that* $\beta_V \cdot \beta_{Q_V} \leq \beta_G$*, and* $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$*, where* $\beta_F = \beta_Y \cdot \beta_{Q'}$.

A symbolic functional decomposition has been schematically shown in Fig. 1. To create binary functions G and H it is necessary to encode the Q_V and Q_U symbolic variables using binary variables. The binary encoding of the Q_V and Q_U variables defines the final encoding of the internal states of the FSM.

Example 3 A symbolic functional decomposition of the FSM from Table 2a. For the example FSM from Table 2a the β_{x_i}, β_Q, $\beta_{Q'}$, β_Y and β_F blankets are as follows:

$$\begin{aligned} \beta_{x_1} &= \{\overline{1, 2, 3, 5, 6, 7, 9, 11, 13, 14, 15, 18, 19, 20}; \\ &\quad \overline{1, 3, 4, 5, 6, 8, 10, 12, 13, 16, 17}\}, \\ \beta_{x_2} &= \{\overline{1, 3, 4, 6, 7, 8, 13, 14, 17, 18, 19}; \end{aligned} \tag{13}$$

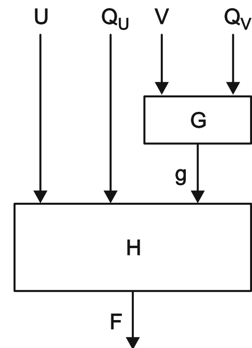


Fig. 1 Schematic representation of symbolic functional decomposition

Table 2 (a) state transition table of the FSM, (b) symbolic encoding of the Q variable

(a)									(b)	
	x_1	x_2	x_3	x_4	Q	Q'	y_1	y_2	Q_U	Q_V
1	–	–	0	0	init0	init1	0	0	u_1	v_1
2	0	1	0	0	init1	init1	0	0	u_2	v_1
3	–	–	1	–	init1	init2	1	0	u_2	v_1
4	1	–	1	0	init2	init4	1	0	u_2	v_2
5	–	1	1	1	init4	init4	1	0	u_3	v_4
6	–	–	0	1	init4	IOWait	0	1	u_3	v_4
7	0	0	0	–	IOWait	IOWait	0	1	u_4	v_3
8	1	0	0	–	IOWait	init1	0	1	u_4	v_3
9	0	1	1	0	IOWait	read0	0	0	u_4	v_3
10	1	1	0	0	IOWait	write0	1	1	u_4	v_3
11	0	1	1	1	IOWait	RMACK	1	1	u_4	v_3
12	1	1	0	1	IOWait	WMACK	0	0	u_4	v_3
13	–	0	1	–	IOWait	init2	0	1	u_4	v_3
14	0	0	1	0	RMACK	RMACK	1	1	u_1	v_4
15	0	1	1	1	RMACK	read0	0	0	u_1	v_4
16	1	1	0	0	WMACK	WMACK	0	0	u_1	v_2
17	1	0	0	1	WMACK	write0	0	1	u_1	v_2
18	0	0	0	1	read0	read1	1	1	u_2	v_4
19	0	0	1	0	read1	IOWait	0	1	u_3	v_2
20	0	1	0	0	write0	IOWait	0	1	u_3	v_1

$$\overline{1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 15, 16, 20}\}, \tag{14}$$

$$\beta_{x_3} = \{\overline{1, 2, 6, 7, 8, 10, 12, 16, 17, 18, 20}; \overline{3, 4, 5, 9, 11, 13, 14, 15, 19}\}, \tag{15}$$

$$\beta_{x_4} = \{\overline{1, 2, 3, 4, 7, 8, 9, 10, 13, 14, 16, 19, 20}; \overline{3, 5, 6, 7, 8, 11, 12, 13, 15, 17, 18}\}, \tag{16}$$

$$\beta_Q = \{\overline{1}; \overline{2, 3}; \overline{4}; \overline{5, 6}; \overline{7, 8, 9, 10, 11, 12, 13}; \overline{14, 15}; \overline{16, 17}; \overline{18}; \overline{19}; \overline{20}\}, \tag{17}$$

$$\beta_{Q'} = \{\overline{1, 2, 8}; \overline{3, 13}; \overline{4, 5}; \overline{6, 7, 19, 20}; \overline{9, 15}; \overline{10, 17}; \overline{11, 14}; \overline{12, 16}; \overline{18}\}, \tag{18}$$

$$\beta_Y = \{\overline{1, 2, 9, 12, 15, 16}; \overline{6, 7, 8, 13, 17, 19, 20}; \overline{3, 4, 5}; \overline{10, 11, 14, 18}\}, \tag{19}$$

$$\beta_F = \beta_Y \cdot \beta_{Q'} = \{\overline{1, 2}; \overline{3}; \overline{4, 5}; \overline{6, 7, 19, 20}; \overline{8}; \overline{9, 15}; \overline{10}; \overline{11, 14}; \overline{12, 16}; \overline{13}; \overline{17}; \overline{18}\}. \tag{20}$$

Let $U = \{x_1, x_2\}$ and $V = \{x_3, x_4\}$ be subsets of the input variable of the FSM. This means that the β_U and β_V blankets are $\beta_U = \beta_{x_1} \cdot \beta_{x_2}$ and $\beta_V = \beta_{x_3} \cdot \beta_{x_4}$:

$$\beta_U = \{\overline{1, 2, 3, 5, 6, 9, 11, 15, 20}; \overline{1, 3, 4, 5, 6, 10, 12, 16}; \overline{1, 3, 4, 6, 8, 13, 17}; \overline{1, 3, 6, 7, 13, 14, 18, 19}\}, \quad (21)$$

$$\beta_V = \{\overline{1, 2, 7, 8, 10, 16, 20}; \overline{3, 4, 9, 13, 14, 19}; \overline{3, 5, 11, 13, 15}; \overline{6, 7, 8, 12, 17, 18}\}. \quad (22)$$

After the β_V and β_U blankets are computed it is possible to construct the β_{Q_U} blanket, which introduces the Q_U symbolic variable and partially encodes the state variable of the FSM. The Q_U variable is one of the inputs of the H block (Fig. 1).

The β_{Q_U} blanket constructed at this stage must satisfy the $\beta_Q \leq \beta_{Q_U}$ condition (β_{Q_U} cannot introduce separations between symbols not separated by β_Q). Additionally, according to Theorem 1, the $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$ condition must be met. This means that the β_{Q_U} blanket (along with β_U) determines how many separations not introduced by $\beta_U \cdot \beta_{Q_U}$, but required by β_F, will have to be provided by β_G.

The β_{Q_U} blanket can be constructed by merging blocks of the β_Q blanket, which guarantees the fulfillment of the $\beta_Q \leq \beta_{Q_U}$ condition. For example, merging the first, sixth and seventh blocks of the β_Q blanket, then the second, third and eighth, and finally merging the fourth, ninth and tenth block yields the following four-block β_{Q_U} blanket:

$$\beta_{Q_U} = \{\overline{1, 14, 15, 16, 17}; \overline{2, 3, 4, 18}; \overline{5, 6, 19, 20}; \overline{7, 8, 9, 10, 11, 12, 13}\}. \quad (23)$$

The Q_U multi-valued variable corresponding to that blanket will have four values: u_1, u_2, u_3 and u_4 (Table 2b).

The next step is the construction of the blanket induced by the output of the G block (the β_G blanket). It must satisfy the $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$ condition, so that together with the blankets induced by the U variable and the Q_U symbolic variable all separations between symbols required by the F function are supplied. The construction of the β_G blanket can be performed using a method known from functional decomposition of combinational functions, based on incompatibility graph coloring. The vertices of this graph are the blocks of the $\beta_V \cdot \beta_Q$ blanket—representing inputs of the G block. The edges of the graph connect the vertices that cannot be merged, because they provide the separations which must be generated by the outputs of the G block (so that the decomposed system can properly implement the F function) and are not supplied by the $\beta_U \cdot \beta_{Q_U}$ blanket. The coloring specifies which blocks of the $\beta_V \cdot \beta_Q$ blanket can be merged to form blocks of the β_G blanket.

$$\beta_U \cdot \beta_{Q_U} = \{\overline{1, 14}; \overline{1, 15}; \overline{1, 16}; \overline{1, 17}; \overline{2, 3}; \overline{3, 4}; \overline{3, 18}; \overline{5, 6, 20}; \overline{6, 19}; \overline{7, 13}; \overline{8, 13}; \overline{9, 11}; \overline{10, 12}\}, \quad (24)$$

$$\beta_V \cdot \beta_Q = \{\overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6}; \overline{7, 8, 10}; \overline{7, 8, 12}; \overline{9, 13}; \overline{11, 13}; \overline{14}; \overline{15}; \overline{16}; \overline{17}; \overline{18}; \overline{19}; \overline{20}\}. \quad (25)$$

For the $\beta_U \cdot \beta_{Q_U}$ and $\beta_V \cdot \beta_Q$ blankets calculated for the example FSM (formulas (24) and (25)) this process yields the following β_G blanket:

$$\beta_G = \{\overline{1, 2, 4, 7, 8, 10, 19, 20}; \overline{3, 7, 8, 12, 16, 17}; \overline{6, 9, 13, 18}; \overline{5, 11, 13, 14, 15}\}. \tag{26}$$

This β_G blanket corresponds to the multi-valued variable with four values, so can be encoded with two binary variables g_1 and g_2. The last stage of the symbolic functional decomposition is the construction of the β_{Q_V} blanket, what is equivalent to the introduction of the Q_V symbolic variable which partially encodes the Q state variable of the FSM. This blanket must satisfy the $\beta_Q \leq \beta_{Q_V}$ and $\beta_{Q_V} \cdot \beta_V \leq \beta_G$ conditions.

The construction of the β_{Q_V} blanket is based on merging the blocks of the β_Q blanket in such a way that the resulting blanket delivers all separations required by the β_G blanket which are not delivered by the β_V blanket. This process can also be based on incompatibility graph coloring. The vertices of the graph are the blocks of the β_Q blanket and the edges connect the vertices which cannot be merged without losing separations important for the G function (separations that have to be delivered at the output of this function and are not delivered by the β_V blanket). For the β_G blanket from (26) and β_Q and β_V blankets respectively from (17) and (22) this process yields the following β_{Q_V} blanket:

$$\beta_{Q_V} = \{\overline{1, 2, 3, 20}; \overline{4, 16, 17, 19}; \overline{7, 8, 9, 10, 11, 12, 13}; \overline{5, 6, 14, 15, 18}\}. \tag{27}$$

Thus, the Q_V multi-valued variable corresponding to this blanket will have four values: v_1, v_2, v_3 and v_4 (Table 2b).

It can be easily verified that the blankets (21)–(23), (26) and (27) satisfy the conditions of the Theorem 1. Thus the example FSM from Table 2a has the symbolic functional decomposition with respect to U, V, Q_U, and Q_V.

The truth table describing the G block was shown in Table 3b. After encoding the β_{Q_V} blanket (of the Q_V symbolic variable) with the q_1 and q_2 binary variables a truth table can be obtained that describes the G Boolean function with four inputs and two outputs (Table 3c). This function can be directly implemented in the logic elements of a typical FPGA architecture. The H function (Table 3a) can be subjected to further symbolic functional decomposition process, until the blocks obtained in the process can be directly implemented in FPGA logic cells.

4 Algorithms

As it was explained previously, the symbolic functional decomposition method comprises a few subsequent stages: the partitioning of the binary inputs into the U and V sets, the construction of the β_{Q_U} blanket, the construction of the β_G blanket and the construction of the β_{Q_V} blanket. After completing these stages the symbolic function F describing the state transition table of the finite state machine can be

Table 3 Blocks of symbolic functional decomposition of FSM from Table 2: (a) H block, (b) G block, (c) G block with binary encoded symbolic variable Q_V

(a)										(b)					(c)					
	x_1	x_2	g_1	g_2	Q_U	Q'_U	Q'_V	y_1	y_2	x_3	x_4	Q_V	g_1	g_2	x_3	x_4	q_1	q_2	g_1	g_2
1	–	–	0	0	u_1	u_2	v_1	0	0	0	0	v_1	0	0	0	0	0	0	0	0
2	0	1	0	0	u_2	u_2	v_1	0	0	1	–	v_1	0	1	1	–	0	0	0	1
3	–	–	0	1	u_2	u_2	v_2	1	0	0	–	v_2	0	1	0	–	0	1	0	1
4	1	–	0	0	u_2	u_3	v_4	1	0	1	0	v_2	0	0	1	0	0	1	0	0
5	–	1	1	1	u_3	u_3	v_4	1	0	0	0	v_3	0	0	0	0	1	0	0	0
6	–	–	1	0	u_3	u_4	v_3	0	1	0	1	v_3	0	1	0	1	1	0	0	1
7	0	0	0	–	u_4	u_4	v_3	0	1	1	0	v_3	1	0	1	0	1	0	1	0
8	1	0	0	–	u_4	u_2	v_1	0	1	1	1	v_3	1	1	1	1	1	0	1	1
9	0	1	1	0	u_4	u_2	v_4	0	0	0	1	v_4	1	0	0	1	1	1	1	0
10	1	1	0	0	u_4	u_3	v_1	1	1	1	–	v_4	1	1	1	–	1	1	1	1
11	0	1	1	1	u_4	u_1	v_4	1	1											
12	1	1	0	1	u_4	u_1	v_2	0	0											
13	–	0	1	–	u_4	u_2	v_2	0	1											
14	0	0	1	1	u_1	u_1	v_4	1	1											
15	0	1	1	1	u_1	u_2	v_4	0	0											
16	1	1	0	1	u_1	u_1	v_2	0	0											
17	1	0	0	1	u_1	u_3	v_1	0	1											
18	0	0	1	0	u_2	u_3	v_2	1	1											
19	0	0	0	0	u_3	u_4	v_3	0	1											
20	0	1	0	0	u_3	u_4	v_3	0	1											

represented as two functions, G and H, ready to undergo subsequent decomposition processes if needed, up to a point when they can be easily implemented (e.g., they *fit* into an FPGA device LUT cells directly). For each of the stages various algorithms were proposed in [30–35].

4.1 Selection of the U and V Input Sets

The first stage of the symbolic functional decomposition method is the partitioning of X, the set of binary inputs, into the U and V (free and bound) sets—the direct inputs to, respectively, the H and G decomposed functions. In most cases $U \cup V = X$; in rare cases some of the inputs might have no bearing on the outputs of the F function and can be discarded altogether, in which case $U \cup V \subset X$.

If $U \cap V = \emptyset$ (i.e., the G and H functions have no common binary inputs) then the decomposition is *disjoint*. If $U \cap V \neq \emptyset$ then the decomposition is *non-disjoint* and some of the binary inputs are shared between the G and H functions.

Various algorithms for generating the U and V sets have been proposed. The simplest one generates all possible U and V combinations. An example implementation could generate all integers between 0 and $2^{|X|} - 1$ (so all $|X|$-bit integers), and for every such integer i generate a pair of U and V sets where the x_j input goes to the U set if the jth bit of i is 0 and to the V set if it's 1. The U and V pairs where the V set is too large (e.g., where V has more than four elements when targeting FPGA devices with four-input LUT cells) can then be discarded to limit the number of considered decompositions. Such generation of the U and V sets is relatively fast, but the number of U and V pairs grows exponentially with the number of inputs and the order in which the pairs are generated is not related to the relative significance of the inputs.

As the blankets constructed in the process of decomposition must satisfy the $\beta_V \cdot \beta_{Q_V} \leq \beta_G$ and $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$ conditions, the selection of U and V sets—which defines the β_U and β_V blankets—has crucial impact. If the β_U blanket provides most of the separations required by the β_F blanket, the β_G blanket does not have to provide many separations and can be implemented on very few binary inputs to the H function. Similarly, if β_V provides most of the separations required by β_G, the β_{Q_V} blanket can have only a few blocks and can be implemented on a very small number of binary inputs to the G function.

In most cases different FSM inputs provide different amount of information relevant for the functionality of the FSM (some inputs bear a lot of significant information, while others almost—or exactly—none); the relative relevance of the inputs can be represented by the number of separations provided by the given input from all of the separations required by the β_F blanket.

As the existing research concludes, the number of separations required by the β_F blanket and provided by a given input varies greatly from input to input. Inputs with many relevant separations should be selected for the U set (so that the most relevant information passes directly to the H function in as few inputs as possible),

while inputs with few relevant separations should be selected for the V set (so the G function can *compress* the relevant information from these inputs into as few blocks of the β_G blanket as necessary, yielding fewer physical lines implementing β_G).

This selection can be performed by creating a sequence s of the inputs ordered by the number of relevant separations they provide (including the state input, represented in s by $\lceil \log_2(|Q|) \rceil$ elements—i.e., the minimal number of bits required to implement the Q state variable), and subsequently generating all numbers from 0 to $2^{|s|} - 1$. The binary representation of each number can then be used again as a bit mask: elements of the s sequence matching 0 bits are selected for the U set, while elements matching 1 bits are selected for the V set. This yields subsequent U and V input sets that favor selecting inputs providing many relevant separations for the U set before yielding U and V combinations that would put these inputs in the V set.

Example 4 U and V set generation based on number of separations for the benchmark *s27* FSM.

The standard benchmark *s27* FSM has four inputs (x_3, x_2, x_1 and x_0) and six states; its β_F blanket requires 463 separations. β_0 (blanket induced by the x_0 input) provides 270 separations, of which 267 are required by β_F (i.e., relevant). Similarly, β_1 provides 104 separations (88 relevant), β_2 provides 182 separations (180 relevant) and β_3 provides 20 separations (16 relevant). β_Q (blanket induced by the state variable) provides 471 separations, of which 384 are relevant. As the FSM has six states, the minimal number of physical inputs required to implement the state variable equals 3—so it can be said that the state variable provides 128 relevant separations per input required. This leads to the following sequence of inputs sorted by general relevance counts: x_0, x_2, q, q, q, x_1, x_3 (where the q elements are the state variable *placeholders*).

Once this sequence is generated, the U and V sets are constructed similarly to the simple algorithm described above—with the exception that the only numbers considered for input selection are these with the count of 1s in their binary representation equal to the desired number of inputs to the G function; also, the q *placeholder* inputs are dropped from the final U and V sets and any repeating (U, V) pairs are ignored.

For the example benchmark *s27* FSM the input generation sequence is shown in Table 4—this example shows that in successive (U, V) pairs the inputs going into the V set are the least relevant ones (and their combinations); this means that the decomposition process which uses this algorithm will give priority to the U and V sets in which the most relevant inputs are passed into the H function, while the G function is used to *compress* information from the least relevant inputs into fewer physical lines.

A variation of this algorithm can order the inputs by the number of *unique* relevant separations (out of the separations required by β_F) provided by each of the input—such algorithm would favour connecting inputs that provide unique information to the H function (as they would get selected for the U set).

Table 4 General relevance U and V sets for *s27* and a device with three-input LUT cells

Number	Sequence: $x_0, x_2, q, q, q, x_1, x_3$		Discard q		Notes
	U set (bit = 0)	V set (bit = 1)	U set	V set	
0000111_b	x_0, x_2, q, q	q, x_1, x_3	x_0, x_2	x_1, x_3	
0001011_b	x_0, x_2, q, q	q, x_1, x_3	x_0, x_2	x_1, x_3	Discarded as repeating
0001101_b	x_0, x_2, q, x_1	q, q, x_3	x_0, x_2, x_1	x_3	
0001110_b	x_0, x_2, q, x_3	q, q, x_1	x_0, x_2, x_3	x_1	
0010011_b	x_0, x_2, q, q	q, x_1, x_3	x_0, x_2	x_1, x_3	Discarded as repeating
0010101_b	x_0, x_2, q, x_1	q, q, x_3	x_0, x_2, x_1	x_3	Discarded as repeating
0010110_b	x_0, x_2, q, x_3	q, q, x_1	x_0, x_2, x_3	x_1	Discarded as repeating
0011001_b	x_0, x_2, q, x_1	q, q, x_3	x_0, x_2, x_1	x_3	Discarded as repeating
0011010_b	x_0, x_2, q, x_3	q, q, x_1	x_0, x_2, x_3	x_1	Discarded as repeating
0011100_b	x_0, x_2, x_1, x_3	q, q, q	x_0, x_2, x_1, x_3	$\emptyset$	
0100011_b	x_0, q, q, q	x_2, x_1, x_3	x_0	x_2, x_1, x_3	
0100101_b	x_0, q, q, x_1	x_2, q, x_3	x_0, x_1	x_2, x_3	
0100110_b	x_0, q, q, x_3	x_2, q, x_3	x_0, x_3	x_2, x_1	
...	...	...	...	...	

4.2 *Construction of the β_{Q_U} Blanket*

The second stage of the decomposition process is the construction of the β_{Q_U} blanket. This blanket has to satisfy the $\beta_Q \leq \beta_{Q_U}$ and $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$ conditions; i.e., the blocks of the β_{Q_U} blanket must be constructed from the blocks of the β_Q blanket in such a way that each is a—not necessarily proper—superset of one of the blocks of the β_Q blanket; also, β_{Q_U} should provide as many separations required by β_F and not provided by β_U as possible.

The number of blocks in the β_{Q_U} blanket is important for the decomposition's quality; the fewer blocks it has, the fewer physical lines will be required in the final implementation of the FSM (the number of physical lines required to implement this blanket is equal to the base-two logarithm of the number of the blanket's blocks, rounded up). At the same time, the fewer blocks this blanket has, the fewer separations it can provide. All of the separations required by the β_F blanket and not provided by the β_U blanket have to be provided by either β_{Q_U} or β_G; the fewer of them come from β_{Q_U}, the more will have to come from β_G, and the more physical lines will be required to implement β_G (thus reducing the advantage of a 'small' β_{Q_U} blanket)—also note that the physical lines implementing β_G are both the outputs of the G function and

inputs of the H function, so their number impacts the complexity of both G and H functions significantly.

To satisfy the $\beta_Q \leq \beta_{Q_U}$ condition, algorithms for constructing the β_{Q_U} blanket must start with blocks of the β_Q blanket and merge them in a way that provides as many separations relevant for the β_F blanket (and not provided by β_U) as possible. As all of the separations not provided by $\beta_U \cdot \beta_{Q_U}$ and required by β_F need to be provided by β_G, targeting a *small* β_G (one with relatively few blocks) can also be the basis of the algorithm.

One of the algorithms described in [32] creates the β_{Q_U} blanket by merging the blocks of the β_Q blanket in a way that yields β_G blanket with the given, small number of blocks, while also striving for as few blocks of the β_{Q_U} blanket as possible; in particular, a result where $\log_2(|\beta_{Q_U}|) < \log_2(|\beta_Q|)$ means that the β_{Q_U} blanket is implementable on fewer physical lines than required by β_Q.

Another β_{Q_U} construction algorithm creates an incompatibility graph where the vertices represent the blocks of the β_Q blanket, and the edges connect those of the vertices that would lose relevant (required by β_F and not provided by β_U) separations if merged; the edges are labeled with the number of relevant separations lost on the given merge. If the vertices of the graph are then subsequently merged in order of increasing edge labels (starting with vertices not connected by any edge), each merging yields a smaller β_{Q_U} blanket—at the increasing cost in the size of the β_G blanket.

Example 5 β_{Q_U} construction for the edge-labeling algorithm.

For the example finite state machine from Table 2 and $U = \{x_1, x_2\}$, the initial β_{Q_U} graph is constructed as on Fig. 2. The vertices of the graph represent the blocks of the β_Q blanket, while the edges connect those of the vertices/blocks which, at best, shouldn't be merged—if they are, some of the separations are lost and have to be provided by the β_G blanket (the weight of the edge equals the number of separations lost in a given merge).

The algorithm first tries to find a pair of vertices that is not connected (so the represented blocks can be merged *at no cost*); in the example case, the $\overline{19}$ and $\overline{20}$ blocks form such a pair. Once the graph is complete, the algorithm finds the *cheapest* pair to merge and merges it.

The number of binary inputs required for implementation of any given blanket is equal to the base-two logarithm from the number of states (rounded up); thus, the initial number of binary inputs for encoding the ten-block β_{Q_U} blanket would be four. The algorithm merges the blocks until the number of the binary inputs is smaller (so, in this case, until there are at most eight blocks, and, thus, three binary inputs suffice) and then yields the resulting β_{Q_U} blanket so that the corresponding β_G and β_{Q_V} blankets can be constructed. The algorithm then makes the β_{Q_U} blanket smaller again (in this example, merges it down to four blocks). This process is repeated until the blanket is merged down to a single block (which creates a β_{Q_U} blanket that does not provide any separations—but also does not require any physical lines; in this case, all of the separations that must be provided by the state variable are passed through the β_{Q_V} blanket).

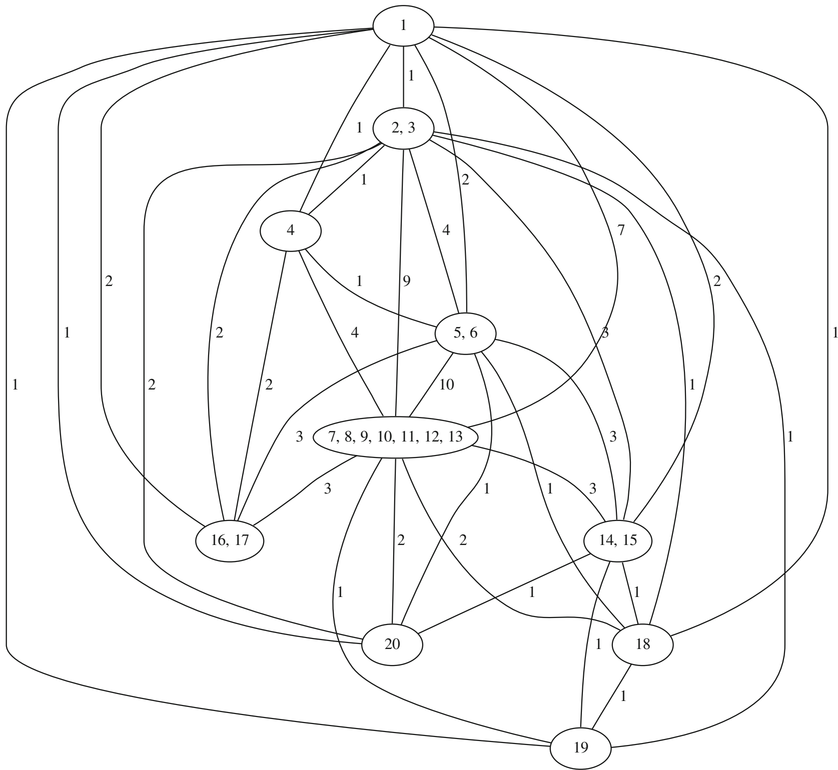


Fig. 2 Incompatibility graph for the β_{Q_U} blanket, unmerged

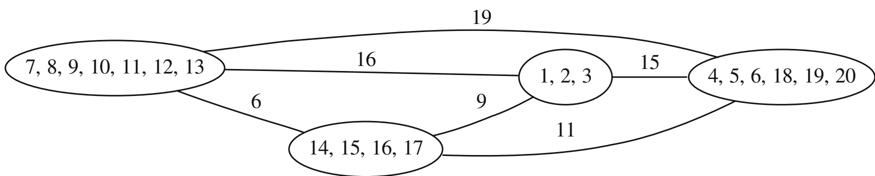


Fig. 3 Incompatibility graph for the β_{Q_U} blanket, merged

In the case of the example incompatibility graph from Fig. 2, the best decomposition was later obtained when the β_{Q_U} blanket was merged down to four blocks, as seen on Fig. 3. In this case the final β_{Q_U} blanket equals

$$\beta_{Q_U} = \{\overline{1,2,3};\ \overline{4,5,6,18,19,20};\ \overline{7,8,9,10,11,12,13};\ \overline{14,15,16,17}\}.$$

4.3 Construction of the β_G and β_{Q_V} Blankets

The last stages of the decomposition process is the construction of the β_G and β_{Q_V} blankets. The β_G blanket has to satisfy the $\beta_U \cdot \beta_{Q_U} \cdot \beta_G \leq \beta_F$ and $\beta_V \cdot \beta_Q \leq \beta_G$ conditions—i.e., it has to provide all of the separations required by β_F and not provided by β_U nor β_{Q_U}, while at the same time its blocks can't be smaller than the blocks of the $\beta_V \cdot \beta_Q$ blanket (this means that the blocks of β_G must be supersets—although not neccessarily proper—of $\beta_V \cdot \beta_Q$ blocks).

The β_{Q_V} blanket has to satisfy the $\beta_Q \leq \beta_{Q_V}$ and $\beta_V \cdot \beta_{Q_V} \leq \beta_G$ conditions: its blocks have to be constructed from the blocks of the β_Q blanket, but in a way that provides the β_G blanket with all the required separations not provided by β_V.

These two blankets can be constructed either separately (first β_G, then β_{Q_V}) or in parallel. When constructed separately, the same algorithms can be used to construct β_G and β_{Q_V}.

One such algorithm, described in [32], is based on graph colouring of an incompatibility graph. An incompatibility graph for β_G construction is one where vertices represent the blocks of the $\beta_V \cdot \beta_Q$ blanket and edges connect those of the blocks that cannot be merged (because doing so would remove separations required by β_F and not provided by $\beta_Q \cdot \beta_{Q_U}$). Giving this graph a proper graph colouring and then creating a blanket by combining blocks represented by same-colour vertices yields β_G that satisfies both conditions.

Similarly, constructing and colouring an incompatibility graph where vertices represent the β_Q blocks and edges connect blocks that must be kept apart to provide separations required by β_G and not provided by β_V yields β_{Q_V} blanket that satisfies both $\beta_Q \leq \beta_{Q_V}$ and $\beta_V \cdot \beta_{Q_V} \leq \beta_G$ conditions.

Another algorithm for creating β_G and β_{Q_V} relies on graph merging; the graphs—created similarly to the ones in the graph colouring algorithm above—have their vertices merged (starting with vertices of the smallest degree). This algorithm is similar to graph colouring in that it yields β_G and β_{Q_V} blankets that efficiently provide required separations, but where graph colouring strives to use as few colours as possible (and so 7-coloured graph is considered better than 8-coloured graph as much as 8-coloured graph is considered better than 9-coloured graph), graph merging recognises that certain merges are more significant than others: a 9-block blanket requires four physical lines to be implemented, while both 8- and 7-block blankets require three physical lines. Being able to merge a 9-vertice graph to 8 vertices is much more important than being able to merge an 8-vertice graph to 7 vertices.

Example 6 β_G and β_{Q_V} construction for the graph merging algorithm.

In the case of the example finite state machine from Table 2 and assuming the example $U = \{x_1, x_2\}$, $V = \{x_3, x_4\}$ once again

$$\beta_Q \cdot \beta_V = \{\overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6}; \overline{7,8,10}; \overline{7,8,12}; \overline{9,13}; \overline{11,13}; \overline{14}; \overline{15}; \overline{16}; \overline{17}; \overline{18}; \overline{19}; \overline{20}\}.$$

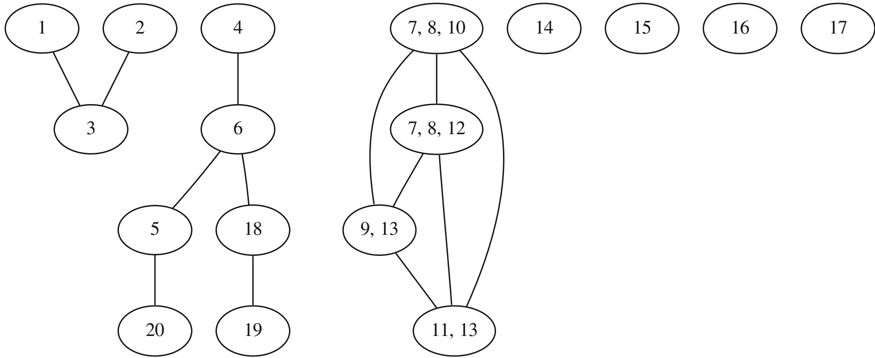


Fig. 4 Incompatibility graph for the β_G blanket, unmerged

Assuming the previously constructed β_{Q_U} blanket

$$\beta_{Q_U} = \{\overline{1,2,3}; \overline{4,5,6,18,19,20}; \overline{7,8,9,10,11,12,13}; \overline{14,15,16,17}\},$$

the initial graph for the β_G blanket is presented in Fig. 4. As can be seen, its vertices represent the blocks of the $\beta_V \cdot \beta_Q$ blanket, while edges connect these of the blocks/vertices which have to be separated.

The number of blocks of this graph governs the number of binary outputs from the G block, and the algorithm merges them until the number of the outputs is smaller than the number of outputs yielded by the initial graph (or until the graph becomes complete, as in this case no more blocks can be merged).

In the case of the above example, the graph is merged down to the four-vertex graph presented in Fig. 5. Thus, the resulting β_G blanket equals

Fig. 5 Incompatibility graph for the β_G blanket, merged

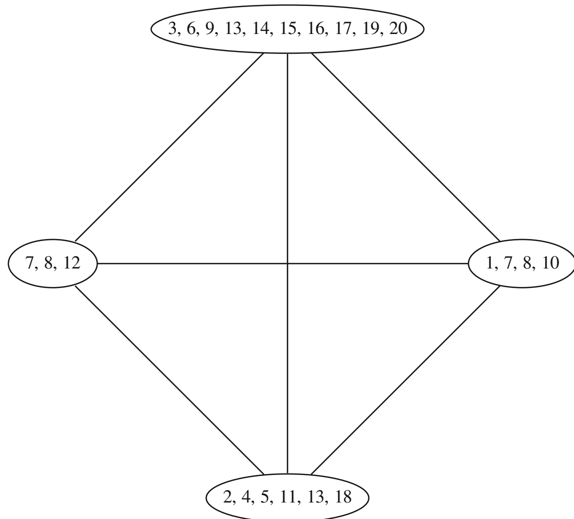
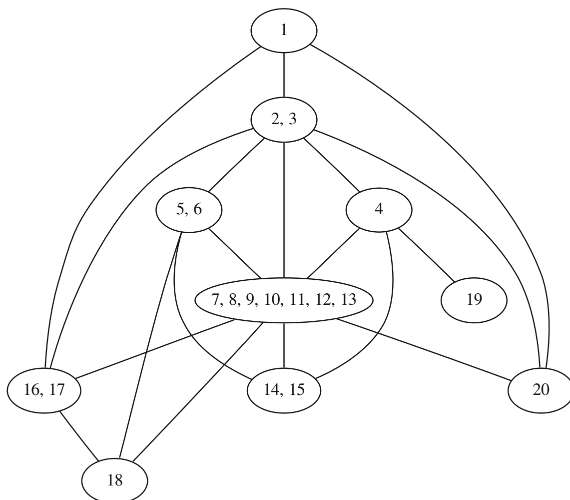


Fig. 6 Incompatibility graph for the β_{Q_V} blanket, unmerged



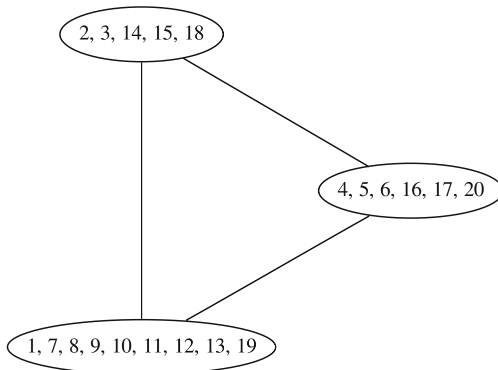
$$\beta_G = \{\overline{1,7,8,10}; \overline{2,4,5,11,13,18}; \overline{3,6,9,13,14,15,16,17,19,20}; \overline{7,8,12}\}.$$

In the case of the example β_G blanket computed above, the initial graph for the β_{Q_V} blanket is presented in Fig. 6.

Again, this graph is merged down until it's implementable with the minimum number of binary inputs; in the case of the example graph from Fig. 6, the resulting graph has three vertices and is presented in Fig. 7. Thus, the resulting β_{Q_V} blanket equals

$$\beta_{Q_V} = \{\overline{1,7,8,9,10,11,12,13,19}; \overline{2,3,14,15,18}; \overline{4,5,6,16,17,20}\}.$$

Fig. 7 Incompatibility graph for the β_{Q_V} blanket, merged



The last algorithm described in [32] constructs β_G and β_{Q_V} in parallel by creating an incompatibility graph from the blocks of the $\beta_Q \cdot \beta_V$ blanket, and then colouring it with two kinds of colours (one for β_G, one for β_{Q_V}) in a way that tries to make both β_G and β_{Q_V} as small as possible in lockstep.

5 Experimental Results

The approach and algorithms described in this chapter were implemented and tested in prototypical software for decomposition of finite state machines.

Table 5 shows decomposition results of standard benchmark finite state machines into four-input, one-output logic blocks obtained using the *art décomp* program and compares them to the synthesis of the same FSMs done with logic design CAD tools A and B. The FSMs were described in Hardware Description Language (HDL) [23], with the states encoded using four different methods—the *JEDI* method described in [22], the *NOVA* method described in [36], the one-hot method (where the number

Table 5 Results of comparison with CAD tools A and B for a device with 4/1 LUT cells

FSM	art décomp	CAD tool A				CAD tool B			
		JEDI	NOVA	Random	One-hot	JEDI	NOVA	Random	One-hot
bbtas	8	9	9	9	9	6	6	6	8
beecount	10	24	24	24	23	23	24	25	51
dk27	5	8	8	8	8	5	5	5	10
donfile	0	54	54	54	54	0	0	0	0
ex3	9	16	16	16	16	17	20	25	42
ex7	10	13	13	13	13	23	23	18	38
lion	3	7	7	7	7	3	3	3	5
lion9	3	16	17	16	17	11	12	20	15
mc	9	10	10	10	10	6	7	7	10
modulo12	0	0	0	0	0	0	0	0	0
opus	24	28	28	28	28	40	49	53	56
s27	7	22	22	22	22	6	15	15	31
s8	0	17	17	17	17	0	0	0	0
shiftreg	4	9	9	9	9	4	3	4	9
train4	3	5	5	5	5	3	3	3	4
train11	5	17	17	17	17	17	18	22	39
$\sum$	100	255	256	255	255	164	188	206	318

of bits used for state encoding equals the number of states, and state number n is encoded with the value 2^n—thus the binary value has always exactly one bit set to 1, and hence the *one-hot* name) and with a random minimal-length encoding. For almost all FSMs the *art décomp* program yields the best results (and they are always better than the ones obtained using the one-hot encoding, which CAD tool B uses by default), and is the best overall solution.

Table 6 presents the same FSMs synthesised from their description in Hardware Description Language (HDL) files and encoded with the same encodings as previously plus all the encodings provided by the CAD tool: sequential encoding, minimal-length encoding and encodings using the Gray and Johnson codes (two approaches where subsequent states are encoded using codes that differ on a single bit). Using a more symbolic HDL representation of a finite state machine yields worse results (except for the one-hot encoding, which yields better overall results when the FSMs are described in HDL). Once again, the *art décomp* program yields by far the best overall results.

Table 7 presents results for example FSMs decomposed into 5/1 and 4/2 blocks and compared with a counterpart implementation using logic design CAD tool A. As with the previous results for this CAD tool, the *donfile* and *s8* static-output FSMs are not recognised properly (albeit implemented in many fewer blocks than in the case of the architecture with 4/1 LUT cells), and again the encoding has only minimal impact on the results. Once again the *art décomp* program yields the best results for every FSM.

Table 8 presents the results of decomposition into 5/1 and 4/2 blocks from the *art décomp* program and compares them with the results published in [18]. Those results were obtained by encoding the states using the *Secode* method (described therein), the *MINISUP* method described in [20] and the previously mentioned *JEDI*, *NOVA* (with two different strategies of obtaining the state encoding), one-hot and random encodings; the encoded FSMs were subsequently synthesised using the *SIS* tool described in [28]. Once again, the *art décomp* program yields the best overall results.

Table 9 present the results of decomposition into 5/1 blocks from the *art décomp* program and compares them with the results published in [29]. Those results were obtained by encoding the states using the *Secode*, *JEDI* and one-hot methods, as well as using the Gray code and simple sequential encoding; the encoded FSMs were subsequently synthesised using the *IRMA2FPGA* tool described in [17]. Here also the *art décomp* program yields the best overall results.

Table 6 Results of comparison with CAD tool B for a device with 4/1 LUT cells

FSM	art décomp	Minimal	Sequential	Gray	JEDI	NOVA	Random	One-hot	Johnson
bbtas	8	6	6	6	6	6	6	9	6
beecount	10	18	18	18	16	18	19	29	33
dk27	5	5	5	5	5	5	5	8	10
donfile	0	0	0	0	0	0	0	47	102
ex3	9	22	28	25	23	31	32	16	31
ex7	10	8	22	14	20	31	29	12	34
lion	3	3	3	3	3	3	3	8	3
lion9	3	16	4	13	15	20	31	20	4
mc	9	6	6	6	6	7	8	10	6
modulo12	0	0	0	0	0	0	0	0	0
opus	24	33	38	31	29	30	38	34	34
s27	7	7	5	10	5	15	14	21	15
s8	0	1	1	1	1	1	1	21	1
shiftreg	4	4	3	4	4	3	4	9	8
train4	3	3	3	3	3	3	3	6	3
train11	5	17	17	25	19	20	29	19	42
$\sum$	100	149	159	164	155	193	222	269	332

Table 7 Results of comparison with CAD tool A for a device with both 5/1 and 4/2 LUT cells

FSM	art décomp	JEDI	Random	NOVA	One-hot
bbara	10	14	14	13	13
bbtas	5	5	5	5	5
beecount	7	13	12	12	12
dk15	7	18	17	18	18
dk17	6	8	9	10	11
dk27	3	5	4	5	5
dk512	7	10	11	11	10
donfile	0	27	27	27	27
ex3	7	9	9	9	9
ex5	5	7	8	7	8
ex6	16	20	20	20	20
ex7	7	7	7	7	7
lion	2	4	4	4	4
lion9	2	10	10	11	11
modulo12	0	0	0	0	0
opus	12	16	16	16	17
s27	4	11	11	11	11
s8	0	9	9	9	9
shiftreg	2	6	6	6	5
train4	2	3	3	3	3
train11	4	9	9	9	9
$\sum$	108	211	211	213	214

Table 8 Results of comparison with *SIS* for a device with both 5/1 and 4/2 LUT cells

FSM	art décomp	Secode	JEDI	NOVA-i	NOVA-io	MINISUP	One-hot
bbara	10	7	11	13	14	16	15
bbtas	5	4	5	3	4	6	6
beecount	7	6	9	8	9	8	12
dk15	7	12	11	13	13	13	17
dk17	6	10	11	9	11	13	17
dk27	3	3	3	4	3	4	6
lion	2	2	2	3	2	2	3
s8	0	1	1	1	1	1	9
$\sum$	40	45	53	54	57	63	85

Table 9 Results of comparison with *IRMA2FPGA* for a device with 5/1 LUT cells

FSM	art décomp	Secode	Gray	JEDI	Sequential	One-hot
bbara	11	12	11	15	15	20
bbtas	5	5	5	5	5	8
beecount	8	6	8	9	10	13
dk15	7	7	7	7	7	12
dk17	6	6	6	6	6	16
dk27	5	5	5	5	5	8
dk512	7	7	7	7	7	19
ex5	5	7	8	11	10	15
ex6	17	21	29	24	29	25
lion	3	3	3	3	3	5
lion9	3	3	4	5	4	10
mc	7	7	7	7	7	7
s27	6	4	5	4	7	15
s8	0	1	6	5	5	8
shiftreg	4	4	4	4	4	9
train11	4	3	7	6	7	13
$\sum$	98	101	122	123	131	203

6 Conclusions

As mentioned in the introduction, the currently widespread two-step approach to implementation of finite state machines in FPGA devices does not yield optimal results due to the definitive character of assigning final binary encodings to the FSM's states prior to the mapping step; because of the complexity of the (often, multi-level) logic synthesis process, creating a universal algorithm for pre-encoding the state variable is very hard. The symbolic functional decomposition method, which side-steps the issue by retaining the relevant information about the states of the FSM through the mapping process, yields better results than the two-step approaches; at the same time, the algorithms currently implementing this method still leaves room for improvement.

References

1. Adamski, M., Karatkevich, A., & Węgrzyn, M. (eds.) (2005). *Design of Embedded Control Systems*. New York: Springer.
2. Armstrong, D. B. (1962). On the efficient assignment of internal codes to sequential machines. *IRE Transactions on Electronic Computers, EC*, *11*(5), 611–622.

3. Ashar, P., Devadas, S., & Newton, A. R. (1989). Optimum and heuristic algorithms for finite state machine decomposition and partitioning. In *1989 IEEE International Conference on Computer-Aided Design. ICCAD-89. Digest of Technical Papers* (pp. 216–219).
4. Ashar, P., Devadas, S., & Newton, A. R. (1990). A unified approach to the decomposition and re-decomposition of sequential machines. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (pp. 601–606).
5. Ashar, P., Devadas, S., & Newton, A. R. (1992). *Sequential logic synthesis. VLSI, computer architecture, and digital signal processing.*, Kluwer international series in engineering and computer science Boston: Kluwer Academic Publishers.
6. Astola, J. T., & Stanković, R. S. (2006). *Fundamentals of switching theory and logic design: A hands on approach*. London: Springer.
7. Brayton, R. K., Rudell, R., Sangiovanni-Vincentelli, A., & Wang, A. R. (1987). MIS: a multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *6*(6), 1062–1081.
8. Brzozowski, J. A., & Łuba, T. (2003). Decomposition of boolean functions specified by cubes. *Journal of Multiple-Valued Logic and Soft Computing*, *9*, 377–417.
9. Chang, S.-C., Marek-Sadowska, M., & Hwang, T. T. (1996). Technology mapping for TLU FPGAs based on decomposition of binary decision diagrams. *IEEE Transactions onComputer-Aided Design of Integrated Circuits and Systems*, *15*(10), 1226–1236.
10. De Micheli, G., Brayton, R. K., & Sangiovanni-Vincentelli, A. (1985). Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *4*(3), 269–285.
11. Devadas, S., Hi-Keung, Ma., Newton, A. R., & Sangiovanni-Vincentelli, A. (1988). MUSTANG: state assignment of finite state machines targeting multilevel logic implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *7*(12), 1290–1300.
12. Dolotta, T. A., & McCluskey, E. J. (1964). The coding of internal states of sequential circuits. *IEEE Transactions on Electronic Computers EC*, *13*(5), 549–562.
13. Du, X., Hachtel, G., Lin, B., & Newton, A. R. (1991). MUSE: a multilevel symbolic encoding algorithm for state assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *10*(1), 28–38.
14. Geiger, M., & Muller-Wipperfurth, T. (1991). FSM decomposition revisited: algebraic structure theory applied to MCNC benchmark FSMs. In *28th ACM/IEEE Design Automation Conference* (pp. 182–185).
15. Hartmanis, J., & Stearns, R. E. (1966). *Algebraic structure theory of sequential machines.*, Prentice-Hall international series in applied mathematics Englewood Cliffs: Prentice-Hall.
16. Humphrey, W. S. (1958). *Switching circuits with computer applications*. New York: McGraw-Hill.
17. Jóźwiak, L., & Chojnacki, A. (2003). Effective and efficient FPGA synthesis through general functional decomposition. *Journal of Systems Architecture*, *49*(4–6), 247–265.
18. Jóźwiak, L., & Ślusarczyk, A. (2000). A new state assignment method targeting FPGA implementations. In *Proceedings of the 26th Euromicro Conference* (Vol. 1, pp. 50–59).
19. Jóźwiak, L., Ślusarczyk, A., & Chojnacki, A. (2003). Fast and compact sequential circuits for the FPGA-based reconfigurable systems. *Journal of Systems Architecture*, *49*(4–6), 227–246.
20. Lemberski, I. (1998). Modified approach to automata state encoding for LUT FPGA implementation. In *Proceedings of the 24th Euromicro Conference* (Vol. 1, pp. 196–199).
21. Lin, B., & Newton, A. R. (1989). Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the IFIP International Conference on VLSI* (pp. 187–196).
22. Lin, B., & Newton, A. R. (1989). Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration* (pp. 187–196).
23. Lipsett, R., Ussery, C., & Schaefer, C. (1989). *VHDL: Hardware description and design*. Boston: Kluwer Academic Publishers.

24. Łuba, T., Rawski, M., Tomaszewicz, P., & Zbierzchowski, B. (2008). Programowalne układy przetwarzania sygnałów i informacji. *Wydawnictwa Komunikacji i Łączności*.
25. Rawski, M. (2004). The novel approach to FSM synthesis targeted FPGA architectures. In *Proceedings of IFAC Workshop on Programmable Devices and Systems, PDS, IFAC* (pp. 169–174).
26. Rawski, M., Jóźwiak, L., & Łuba, T. (2001). Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures. *Journal of Systems Architecture*, *47*, 137–155. Elsevier Science B.V.
27. Scholl, C. (2001). *Functional decomposition with application to FPGA synthesis*. Boston: Kluwer Academic Publisher.
28. Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., et al. (1992). *SIS: A system for sequential circuit synthesis*. Berkeley: University of California.
29. Ślusarczyk, A. (2004). *Decomposition and encoding of finite state machines for FPGA implementation*. Eindhoven: Technische Universiteit Eindhoven.
30. Szotkowski, P. (2008). A comparison of symbolic functional decomposition algorithms for finite state machine implementation in FPGA devices. *III Konferencja naukowo-techniczna doktorantów i młodych naukowców* (pp. 381–385).
31. Szotkowski, P. (2009). Input selection methods for symbolic functional decomposition of finite state machines. In *Proceedings of the 4th International PhD Students and Young Scientists Conference* (pp. 362–367).
32. Szotkowski, P. (2010). Symbolic functional decomposition method for implementation of finite state machines in FPGA Devices. *PhD thesis*. Politechnika Warszawska.
33. Szotkowski, P. & Rawski, M. (2007). Symbolic functional decomposition algorithm for FSM implementation. In *The International Conference on "Computer as a Tool" EUROCON* (pp. 484–488).
34. Szotkowski, P. & Rawski, M. (2008). A graph-based symbolic functional decomposition algorithm for FSM implementation. In *2008 Conference on Human System Interactions* (pp. 34–39).
35. Szotkowski, P., Rawski, M., & Selvaraj, H. (2009). A graph-based approach to symbolic functional decomposition of finite state machines. *Systems Science*, *35*(2), 41–47.
36. Villa, T., & Sangiovanni-Vincentelli, A. (1990). NOVA: State assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *9*(9), 905–924.

Using UML Behavior Diagrams for Graphical Specification of Programs for Logic Controllers

Grzegorz Bazydło and Marian Adamski

Abstract The Unified Modeling Language (UML) is one of the most popular software engineering standards. The UML was designed for specifying, visualizing, constructing and documenting artifacts of software systems. But it could be also very useful for business modeling and can be used successfully for modeling digital systems, including logic controllers. The current version of UML contains fourteen types of diagrams. These diagrams help designer to model large and complex systems, but not all of the diagrams can be suitable for use in the area of hardware design. In the chapter each type of behavior diagrams is analysed and illustrated as an appropriate example. The results show which types of diagrams can be useful in the digital system design process.

Keywords UML · Behavior diagram · State machine · Logic controller

1 Introduction

Nowadays the design process of the digital system can be long and complicated. Before a digital circuit goes into production, the designer must go through all the phases of digital system design [1]:

- specification,
- verification of the specification,
- synthesis,
- verification of the implementation,

G. Bazydło (✉)
Institute of Electrical Engineering, University of Zielona Góra, ul. prof. Z. Szafrana 2, 65-516 Zielona Góra, Poland
e-mail: g.bazydlo@iee.uz.zgora.pl

M. Adamski
Institute of Metrology, Electronics and Computer Science, University of Zielona Góra, ul. prof. Z. Szafrana 2, 65-516 Zielona Góra, Poland
e-mail: m.adamski@imei.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*, Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_10

- documenting the project,
- preparation of test.

The most important phase of the design process is the specification stage, because it has the greatest influence on the quality of the results. Wrong specification generates always an erroneous system. The user requirements are the basis for the development of system specification, and are given very often informally in natural language. Therefore the main aim of this phase is to precisely express the external effects of the designed system. An additional advantage of the formal specification is ensuring that the system behaves in a way acceptable by the future user.

Because nowadays the modelled systems are large and complex, the designer has to use special software for specification, simulation, synthesis and implementation. The main goal is to produce more efficient and faultless systems, because the correction of hardware errors can be very expensive and is often associated with replacement of the device. Therefore the goal of using advanced languages (e.g. UML), methods and tools is to eliminate as many errors at the specification phase as it is possible.

2 Unified Modeling Language

The Unified Modeling Language (UML) is one of the better known modeling languages of software systems, especially object-oriented. The creators of UML are three methodologists: Grady Booch, Ivar Jacobson and James Rumbaugh [4]. The first version (0.8) of UML was published in 1995, and the current version is 2.5 [11]. The main aim of the UML is to support the specification, visualization, construction, and documentation of software systems [4]. It is possible, thanks to 14 types of diagrams, to allow the designer to look at the modelled system from different perspectives, with the emphasis on certain aspects. UML diagrams can be used not only for communication between project team members, but also in conversation with the end user or customer. The UML supports concurrent, hierarchical and distributed systems, and can be used to present system details to meet the requirements or make analysis [6].

Diagrams—the main tool offered by the UML—can be divided into two groups: structure and behavior diagrams. Seven types of the diagrams (class, object, package, component, composite structure, profile and deployment) are used mainly in the presentation of the physical organization of the elements in the system. Behavior diagrams, in contrast to the structure diagrams, focus on describing the behavior of the system components. They are very helpful in describing requirements, operations and internal state changes. Members of this group are as follows: use case, activity, state machine, sequence, communication, interaction overview and timing diagrams.

UML may be advantageous at some stages of the digital systems design process, although it was created primarily to support engineers in the software design process. In the following sections, with the use of an exemplary control system, we will analyse each UML behavior diagram to show at which stage and how it can be used. The possibility and advantages of using UML structure diagrams was discussed in detail in [3].

3 Logic Controller Example

Figure 1 presents a mixer machine for beverage production and distribution (Mixer). The example is taken from [12] and supplemented with support for system failure [2]. The detailed description of the behavior of the Mixer control system can be found in [3]. To better understand the rest of the chapter, we decided to cite it.

The controller works in the following way: the operator pressing the start button (*x1*) initiates the processes in which tanks 1 and 2 are being filled and the containers for the beverages are delivered (signal *y3*). Active signal *x4* means, that containers have been placed correctly on the trolley. Then valves *y10* and *y11* are opened until the tanks are filled, and this information is indicated respectively by sensors *x5* and *x7*. In turn, the delivery of the containers is connected with the movement of the trolley with containers (active signal *y12*) and finishes when the trolley reaches the sensor *x13*. After filling of the tanks, the ingredients are being prepared, which is initiated with signals *y1* and *y2*. An indication of the sensors: *x2* for the first container and *x3* for the second container means that the ingredients in tanks 1 and 2 were prepared. Ready components are poured into the third tank by opening valves *y5* and *y6* and mixed (active signal *y4* until deactivation *x9*). The valves are closed after emptying the tanks 1 and 2. The situation is signaled by sensors *x6*, *x8* and *x9* respectively. When one of the containers is ready, it is independently filled and closed (signals *x10* and *x11*). After the completion of both processes, the trolley with containers moves

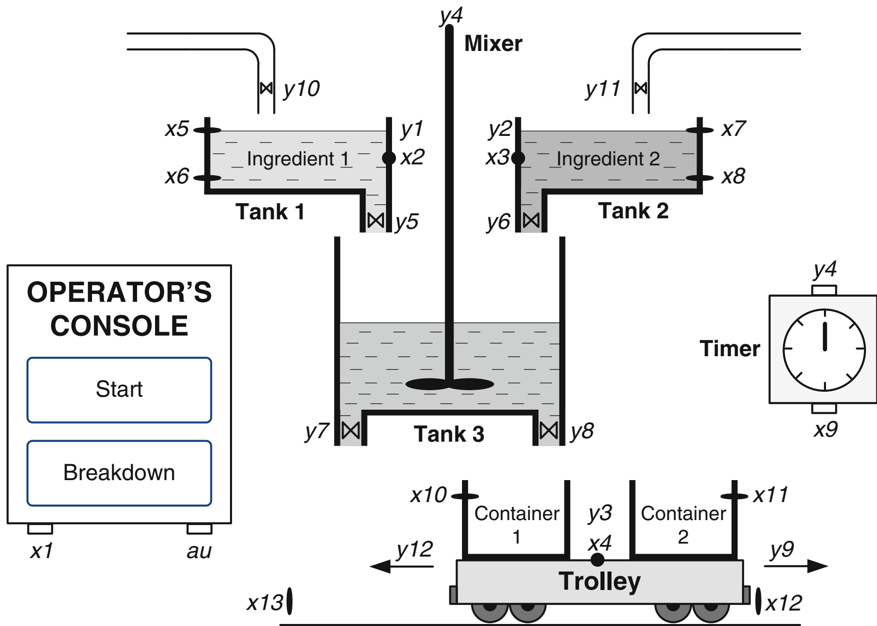


Fig. 1 Industrial Mixer process diagram

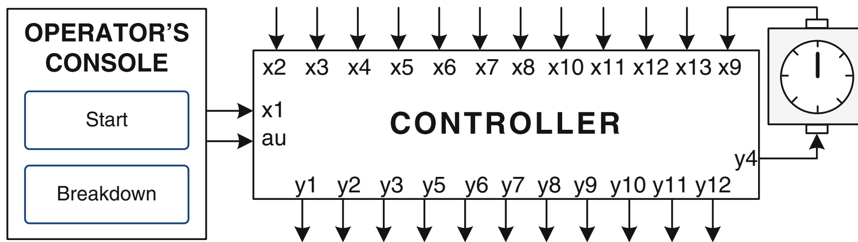


Fig. 2 A controller's block diagram for Mixer control system

back to its initial position, which is signaled by *y9*. Sensor *x12* is active when the system is ready for the further operation [3].

Formally, the presented controller consists of fourteen input sensors and twelve output signals. Controller's block diagram is shown in Fig. 2.

3.1 Use Case Diagram

The first step in the design system process is the analysis of the user requirements [15]. Use case diagrams can be very helpful here, because they represent the functional requirements of the system and provide implementation-independent view of what is expected of the system [10]. Use case diagram show the features of the designed system in a way seen by the future users. A user symbol used on the diagrams, called "an actor", represents the master system that will be used. In this particular case it is a person—namely the system operator. Ovals with names in the middle represent the "use cases". Between use cases several types of associations can occur. The two most common are: include and extend [4].

In the digital system design, especially logic controllers design, use cases play a different role than in the object-oriented systems design. There use cases represent not features of the design system, but rather functions of the controlled object [8]. Figure 3 shows the use case diagram for the Mixer control system example.

Use case diagrams can be very useful at the specification phase of logic controllers design, because they allow to model the interface of the system [7]. The use cases presented in Fig. 3 overlap with the operation modes of the controller, which is also reflected in the operator's console (two buttons for the two main use cases). In addition, this diagram does not require the user to possess subject-specific engineering knowledge and is therefore very helpful in discussing with the customer about the requirements [2].

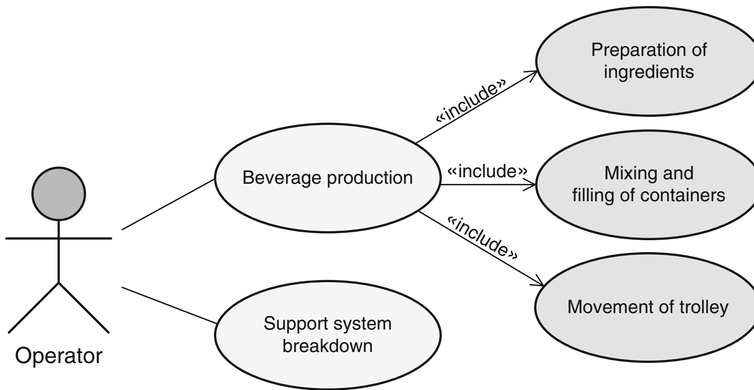


Fig. 3 A use case diagram for the Mixer control system

3.2 *Activity Diagram*

An activity diagram can be seen as one of the type of a block diagram. It shows a control flow from one activity to another. Activity is a kind of behavior consisting of at least one indivisible action. When an activity ends it means, that the system state has changed, and control is immediately passed to the next activity. To describe the control flow, it is convenient to use a "token" as a term, an idea taken from the Petri nets [13].

Activity diagrams can also be used to model the control flow inside the use case [4, 6]. In Fig. 4 an activity diagram is shown realizing the control flow from *Beverage production* use case (Fig. 3) for the Mixer control system.

Activity diagrams can be very helpful in the design of test programs (so-called "testbenches")—used at the stage of behavioral simulation—because they show the selected control scenarios.

3.3 *State Machine Diagram*

State machine diagrams are used to show the control flow between the object states. A prototype for the state machine diagrams were statecharts [5]. The state models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur or it can model dynamic conditions such as the process of performing some behavior [11]. The state machine is the sequence of object's states reached in response to input events and reactions to these events. Hence, the diagram shows the state machine of one object or the whole system treated as single entity [4]. By the use of the composite states it is possible to model the system at the given level of abstraction.

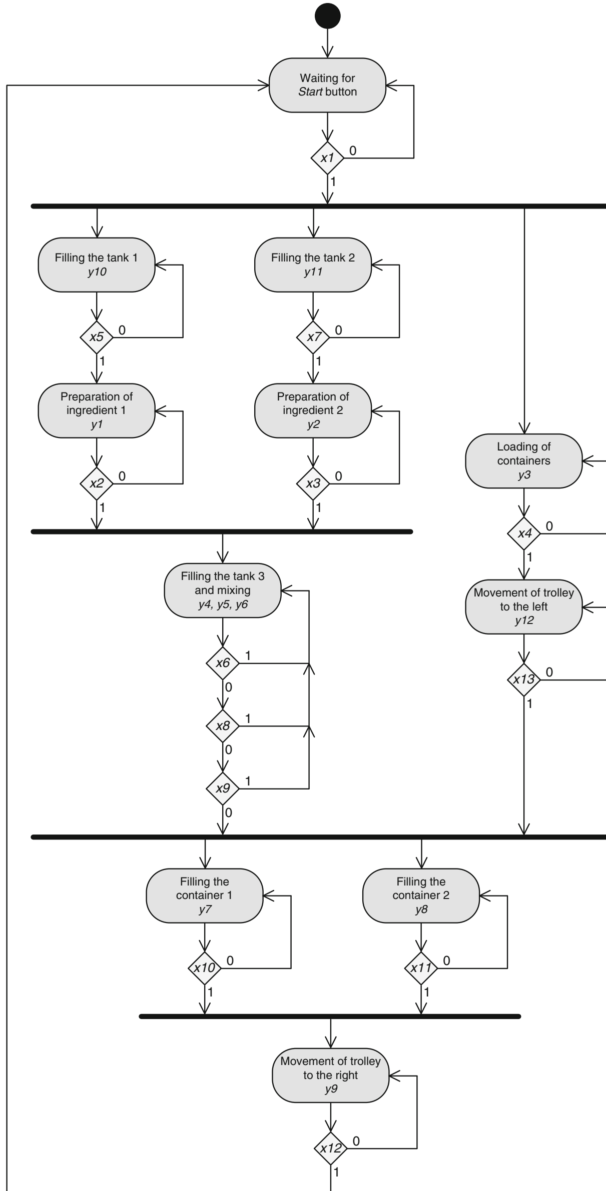


Fig. 4 An activity diagram for the *Beverage production* use case (from Fig. 3)

Figure 5 presents the state machine diagram for the Mixer control system example at the highest level of hierarchy. Note that the states *Beverage preparation and movement of trolley to the left* and *Filling of containers* are composite states with hidden details (marked with two connected circles at the bottom right corner).

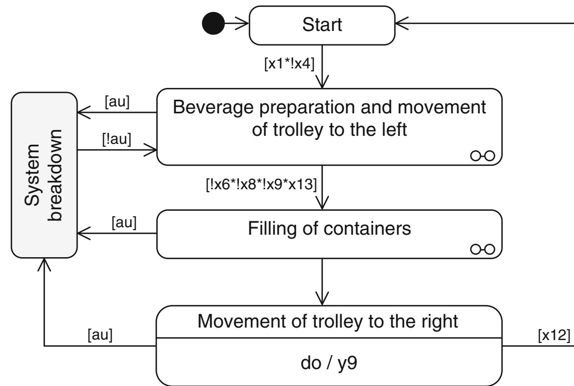


Fig. 5 A state machine diagram for the Mixer control system

Undoubtedly the state machine diagrams are the most important and the most effective tool for the graphic specification of programs for logic controllers. With the support of concurrency and hierarchy of the design system and the precise semantics, state machine diagram allows one to develop a complete and unambiguous description of the logic controller behavior [8, 9, 14]. The diagram represents all states in which the controller object can be, and therefore it can be also helpful in the development of test scenarios. A detailed description of the state machine diagrams for specification of logic controllers' programs can be found in [2].

3.4 Sequence Diagram

The other four types of diagrams (sequence, communication, interaction overview and timing diagrams) belong to the "interaction diagrams" subgroup. Most commonly used diagrams here are the sequence diagrams. They are very helpful in highlight the sequence of messages transmitted between system elements (usually objects) during system's work. Usually, the sequence diagram shows the behavior of the system for only one use case [10]. Sequence diagram has two dimensions: objects (horizontal) and time (vertical). The arrows indicate the messages sent or the induced operations. An example sequence diagram that realizes *Movement of trolley* use case (from Fig. 3) is presented in Fig. 6.

In Fig. 6 the object *controller* is the only element of the system related to the logic controller program (as it was mentioned above, such programs are not usually object-oriented). Other objects represent real nodes, and positioning them on the diagram is rather unnatural, although it may help in understanding the behavior of the controller. Therefore sequence diagrams are in our opinion not very useful in the specification of programs for logic controllers. However, some authors indicate that the diagrams can be used to develop test scenarios for designed system [8].

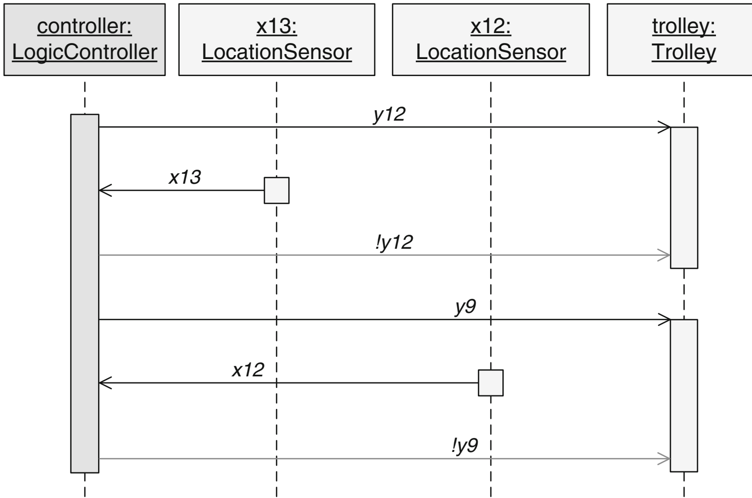


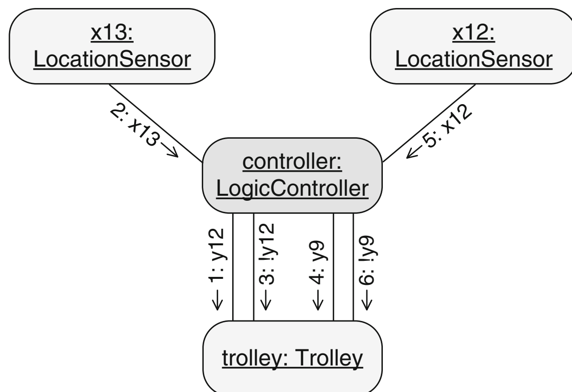
Fig. 6 A sequence diagram for the *Movement of trolley* use case (from Fig. 3)

3.5 Communication Diagram

Communication diagrams are very similar to sequence diagrams, the difference lies in the way of presenting the communication between system elements. The communication diagrams place greater emphasis on the objects rather than on the sequence of the exchanged messages. Figure 7 shows the communication diagram that realizes *Movement of trolley* use case from Fig. 3.

Communication diagrams similarly to sequence diagrams, due to their strong object-oriented nature, are in our opinion not very useful in designing programs for logic controllers.

Fig. 7 A communication diagram for the *Movement of trolley* use case (from Fig. 3)



3.6 Interaction Overview Diagram

Interaction overview diagrams are not commonly used in software or non-software system design. The interaction overview diagram is a combination of activity diagram and sequence diagrams, in which activities are replaced with sequence diagrams. An exemplary interaction overview diagram for the Mixer control system that realizes *Movement of trolley* use case from Fig. 3 is shown in Fig. 8.

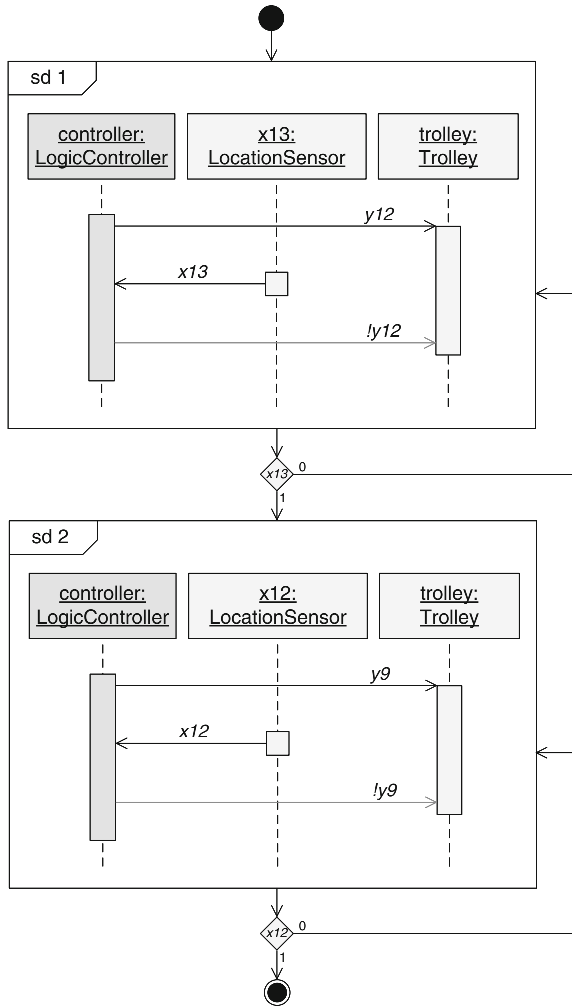


Fig. 8 An interaction overview diagram for the *Movement of trolley* use case (from Fig. 3)

Due to object-oriented nature of sequence diagrams, which are the main components of interaction overview diagrams, the use of such diagrams in the specification programs of logic controllers is rather doubtful.

3.7 Timing Diagram

These diagrams emphasize the time constraints of a single element or an entire group of objects. It is worth mentioning that this type of a diagram was added to UML only in version 2.0 and above, and therefore their use in the design of digital systems is not yet thoroughly investigated. The timing diagrams are very similar to the graphs generated during the system simulation, at the "verification of the specification" stage (see Sect. 1). This similarity can be a rationale for further research into the automatic comparison of the expected results with the results obtained during the system simulation or test programs (testbenches) generation. An exemplary timing diagram for the Mixer control system is shown in Fig. 9.

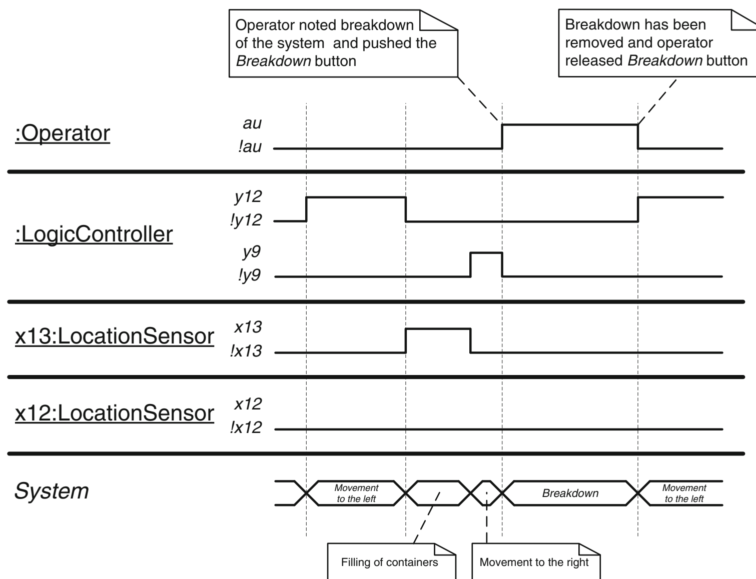


Fig. 9 A timing overview diagram for the Mixer control system

4 Conclusions

Unified Modeling Language is a set of graphical notations used to illustrate, specify, construct and document the modeled systems, especially the object-oriented. Appropriate diagrams support system analysis for the structure or behavior at a given level of abstraction. However, not all types of diagrams are suitable for use in the design of digital systems (this is especially true in the case of structure diagrams [3]), since it was not the main rationale for their development. The main tools for modeling programs for logic controllers are rather behavior diagrams. The most important and popular diagrams for modeling the behavior of logic controllers are the state machine diagrams and mainstream researchers focus on creating a complete and unambiguous specification of a complex digital system using this kind of diagram (or one of its variants) [2, 5, 8, 9, 14].

We also see the possibility of using activity and use cases diagrams. Activity diagrams show the flow of the system control in a probable scenario and can be very useful in generating the test programs. Use case diagrams are used to analyse the needs of the modeled system and allow one to specify, for example, the controller modes and can support the development of a system interface. Both types of diagrams complement the system behavior description expressed with state machine diagrams. It is worth mentioning that the timing diagrams can be very interesting in terms of their use for verification the results obtained during the system simulation. Other behavior diagrams play a minor, supporting role, enabling a fuller analysis of the system, a more accurate verification or are used just to document the project. In some

Table 1 The role of the UML behavior diagrams in logic controllers design process

Design process steps	UML
Specification	State machine diagram
Verification of the specification	State machine diagram Use case diagram Activity diagram Timing diagram
Synthesis	–
Verification of the implementation	–
Documenting the project	Use case diagram Activity diagram State machine diagram Sequence diagram Communication diagram Interaction overview diagram Timing diagram
Test preparation	Use case diagram Activity diagram State machine diagram Sequence diagram Timing diagram

cases the use of object-oriented diagrams to present the real system elements (nodes) seems an unnatural solution. However, without objects unrelated to the controller program, the diagram (e.g. sequence, communication diagram) would simplify to a single element. It follows from the fundamental difference between software design and control algorithm design. In the former, the aim is to create a design based on the analysis of an object-oriented data model, while in the latter the aim is to develop a behavior model. There is no need to perform object analysis, since objects are already directly defined [8].

Table 1 summarizes the UML behavior diagrams in terms of their support of the digital controllers design process (see Sect. 1).

References

1. Adamski, M. (1990). *Design of digital circuits with the use of the systematic structural method*. Zielona Góra: Wydawnictwo Wyższej Szkoły Inżynierskiej. (in Polish).
2. Bazydło, G. (2011). *Graphic specification of programs for reconfigurable logic controllers using unified modeling language* (Vol. 19)., Lecture notes in control and computer science Zielona Góra: University of Zielona Góra Press.
3. Bazydło, G. (2014). Using UML structure diagrams for graphical specification of programs for logic controllers. In A. Bukowiec, G. Borowik, & M. Doligalski (Eds.), *New trends in digital systems design* (Vol. 836, pp. 68–80)., Fortschritt-Berichte VDI Dusseldorf: VDI Verlag GmbH.
4. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The unified modeling language user guide* (2nd ed.). Reading: Addison-Wesley Professional.
5. Drusinsky, D., & Harel, D. (1989). Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *8*(7), 798–807.
6. Fowler, M. (2005). *UML 2.0 Distilled*. Warsaw: Oficyna Wydawnicza LTP. (in Polish).
7. Gomes, L., & Costa, A. (2003). From use cases to system implementation: statechart based co-design. In *Proceedings of the first ACM and IEEE international conference on formal methods and models for co-design, MEMOCODE'03* (pp. 24–33). IEEE.
8. Łabiak, G., & Adamski, M. (2006). Using UML for modeling discrete control. *Pomiary, Automatyka, Kontrola*, *16*(6), 50–52. (in Polish).
9. McUmber, W., & Cheng, B. (1999). UML-based analysis of embedded systems using a mapping to VHDL. In *Proceedings of the 4th IEEE international symposium on high-assurance systems engineering* (pp. 56–63).
10. Pilone, D., Piwko, Ł., & Pitman, N. (2007). *UML 2.0: Almanac*. Gliwice: Helion. (in Polish).
11. *Unified Modeling Language. Superstructure. v2.5*. Object Management Group (2015). www.omg.org/spec/UML/2.5.
12. Valette, R. (1978). Etude comparative de deux outils de representation: Grafcet et reseau de Petri. *Le Nouvel Automatisme*, *1978*(3), 377–382. (in French).
13. Wiśniewski, R., Stefanowicz, Ł., Bukowiec, A., & Lipiński, J. (2014). Theoretical aspects of Petri nets decomposition based on invariants and hypergraphs. In *Lecture notes in electrical engineering, proceedings of the 8th international conference of multimedia and ubiquitous engineering (MUE), Zhangjiajie, China* (Vol. 308, pp. 371–376).
14. Wood, S., Akehurst, D., Uzenkov, O., Howells, W., & McDonald-Maier, K. (2008). A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. *IEEE Transactions on Computers*, *57*(10), 1357–1371.
15. Wrycza, S., Marcinkowski, B., & Wyrzykowski, K. (2005). *UML 2.0 in modeling software systems*. Gliwice: Helion. (in Polish).

Various Interpretations of Actions of UML Activity Diagrams in Logic Controller Design

Michał Grobelny, Iwona Grobelna and Marian Adamski

Abstract UML activity diagrams in version 2.x can be used as a semi-formal specification technique for logic controller design. The chapter provides various interpretations of activity diagram actions. An action is an elementary indivisible operation in the system which cannot be decomposed. However, it can be treated in different ways—it can be dynamic, state-oriented and with starting and stopping conditions. Each interpretation has its own characteristics and represents another point of view on the designed system.

Keywords Activity diagrams · Design · Logic controllers · UML

1 Introduction

Unified Modelling Language (UML) [8–10] can be used in various application fields, starting from software engineering and ending with hardware specification focusing especially on logic controllers. The most useful in this area are the activity diagrams (especially in UML version 2.x, discussed in this chapter and considered e.g. in [6, 7]) or the state machine diagrams (considered in e.g. [1]).

UML language is easy understandable for a wide range of people, even for non-engineers. It is simple to draw the diagrams and to present them. However, UML

M. Grobelny (✉)
Department of Media and Information Technologies, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: m.grobelny@kmti.uz.zgora.pl

I. Grobelna
Institute of Electrical Engineering, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: i.grobelna@iee.uz.zgora.pl

M. Adamski
Institute of Metrology, Electronics and Computer Science, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: m.adamski@imei.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_11

diagrams are not well supported by formal mechanisms, especially by the formal analysis and verification methods. To avoid this inconvenience and to benefit from a user-friendly specification technique, a UML activity diagram describing logic controller behavior can be transformed into a control Petri net [6, 7], where much more formal methods are available. Bidirectional transformation allows efficient usage of both specification techniques at the same time and can be used to translate existing projects only in a predetermined direction.

Activity diagrams are used to describe the behavior of a designed logic controller. Their basic elements include actions, activities and flows between them, extended by additional nodes, such as fork and join nodes. An activity diagram starts with an initial node and ends with a final node. The elements are combined together to show what a logic controller is supposed to do.

An *action* of a UML activity diagram is *an elementary indivisible operation in the system which cannot be decomposed*. It can be treated in different ways depending on how detailed the specification is. The fact that an action cannot be divided into smaller elements is also very important as it affects essentially, together with action interpretation, the way in which the transformation is performed.

The chapter is structured as follows. Section 2 describes the indivisibility of UML activity diagram actions. Section 3 shows possible different interpretations of an action in order to establish a view perspective on designed system and to enable the transformation into control Petri nets as another specification technique. Finally, Sect. 4 summarizes the chapter.

2 Indivisibility of Actions

An important aspect that should be taken into account, especially during transformation into another specification technique, is the indivisibility of UML activity diagram actions. According to the UML specification [8], actions are indivisible in the context of a particular diagram. They are the smallest logical elements. However, while considering control systems, a single action can be viewed from different levels of abstraction.

If the analysis is performed with the client and the main actions of the system are determined (without going into details), then a sample action (that cannot be divided from the point of view of the diagram) can be a general descriptive action *Close the door* (Fig. 1a), without looking into its structure and action mode.

The process of closing the door itself, even though it is a simple task, may invlove changes of input and output signals. Then, after the prior action has been detailed (see Fig. 1), it can be broken down into three actions relating to signals of the logic controller. Verbal specification for the device will then include actions *Enable door closing*, *Wait for door closing* and *Disable door closing* (Fig. 1b). At this stage, the system designer knows how (roughly speaking) the logic controller is supposed to work.

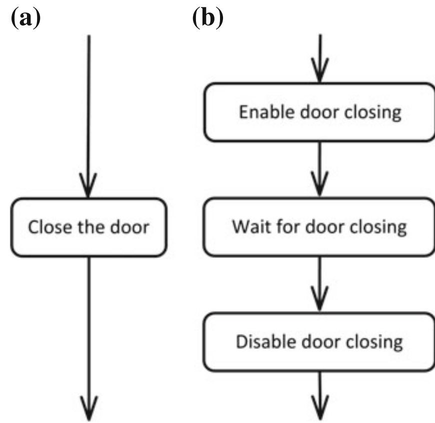


Fig. 1 Two possible representations of one process—using one action (**a**) and three actions (**b**)

On a deeper stage of abstraction, descriptions will be directly converted to signals of a logic controller and will operate on them. For example, the action *Enable door closing* will be interpreted as an assignment of logical truth value to the appropriate output signal $y1 := 1$, so in this case—turning on the actuator responsible for closing the door. Action *Wait for the door closing* will maintain active output signal $y1$ and will continue until the sensors detect the fact that the door has been closed. Next, simple action *Disable door closing* will be interpreted as deactivation of the corresponding output signal, which can be described at the level of signals as the expression $y1 := 0$, which in this case will result in turning off the actuator responsible for closing the door and thus a simple process of closing the door is completed.

It should be noted that in the case of a descriptive diagram at the lowest level of the hierarchy and a signal diagram, the *indivisibility* becomes a real indivisibility, because it is impossible to divide the input signal setting into "finer" actions. Assigning values to signals of a logic controller is a simple operation realized in one clock cycle. In other (previous) cases indivisibility is only from the point of view of the diagram and results from the UML specification. Hence, it can be concluded that the action

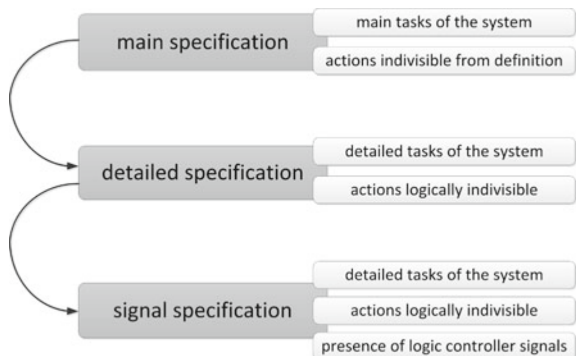


Fig. 2 Indivisibility of action depending on how detailed the specification is

is dependent on the perspective of the problem under consideration, the person who will create the diagram and its context (see Fig. 2).

The above considerations apply to building specifications from the general to the particular (called *top-down*) [2]. However, the described feature of action is also important in case of aggregating actions or building a diagram from the particular to the general (called *bottom-up*) [2]. Then higher-level actions are built from small, logically indivisible ones. They are indivisible only in the context of the diagram (indivisible by definition). Some complex activities can also appear at a higher level diagram in place of an action. They can be used in order to obtain the full hierarchy and the opportunity to develop the activity diagrams using a lower level diagram, which specifies only a particular element.

3 Possible Different Interpretations of an Action

A logic controller is an example of a reactive system, which communicates with the environment through a set of input and output signals and responds to some events. Therefore, the characteristics of actions (performed by the designed device) and their possible interpretation should be considered. It is important to answer the question whether the action is only a short event representing a dynamic change between two states, or maybe it is a lengthy state that is a kind of reaction to the dynamics of the object and required changes or whether it should be considered in a more complex way as the dynamics of a system with a longer duration.

An action of UML activity diagram from the assumptions (by definition) represents the dynamics of the system. It represents a change of transition from one system state into another. The action may, however, also be a representation of the system state. Seemingly contradictory combination of words *action* and *state* may have its logical justification. This is due to the perception of the system and ongoing processes. Action interpretation (and so activity interpretation) is very important in the transformation of UML activity diagrams into control Petri nets. It influences the way and target elements of control Petri nets to which the action will be translated.

There are several aspects that affect the perception of the action and its dynamic or state-oriented characteristics. A consequence of this multiplicity of interpretation is more than one way of mapping the actions and activities of UML language into control Petri nets.

A particularly important aspect when considering the action characteristics is its duration. Simple, elementary and infinitely short system tasks will be perceived only as its dynamics characterizing quick changes between states. Examples of such elementary actions can be changes of signal values, simple arithmetic operations or execution of tasks such as *light the LED*. Considering the context of control systems, simple operations are usually those that are performed in a single clock cycle or within one performing of the main loop. Actions with longer duration are interpreted differently. Such long-term processes are more clearly identified with the system state, as it is easier to imagine this process functionality.

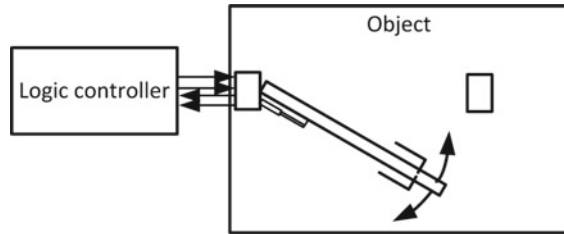


Fig. 3 Schema of system controlling door closing

Another aspect is the perception of the system and its processes. The perspective from which the system is modeled by a designer or analyst is very important. The possibility to view the system from different perspectives is particularly evident in the case of control systems, because then two sides of the system can be considered. The control system consists of two main elements, namely a logic controller and a controlled object (Fig. 3). The analyst or system designer can describe the system from both sides. Depending on the perspective, the interpretation of a particular process and its dynamic or state-oriented character changes.

Differences in interpretation are shown on the seemingly simple and obvious example of the door closing process. This process can be visualized using UML in the form of action *Close the door*. It is assumed that the process is automated and implemented by a logic controller based on the signals from the sensors (e.g. limit switch) using actuators (e.g. servomotor). The logic controller closes the door after the receipt of a signal by appropriately manipulating the servomotor (an example is shown in Fig. 3).

This process can be interpreted both as a state and as an action of the system. In this seemingly contrary definition, the emphasis is put on its interpretation. The dynamic interpretation is oriented only on the change that occurs in the system in relation to the action realization. The state interpretation, beyond mapping changes in the system, refers indirectly to the duration of an action and to the state, in which the designed logic controller is executing an action.

Now, consider the dynamic nature of the action *Close the door*. The door closes and a change, i.e. transition from state *Door open* into state *Door closed*, occurs in the system. Schematic notation of the dynamic nature involving the simple change between two states is shown in Fig. 4a. It is a rapid change causing changes in the controlled object, without paying much attention to the aspect of time. This interpretation brings to mind a transition, which is a component that connects two places corresponding to the successive states of the system. In this case, the transition joins two places corresponding to state *Door open* and *Door closed*.

The second possible interpretation of an action is its identifying with the state of the system. Closing the door in the real world is not the process of infinitely short duration. Thus, looking at it from the point of view of a logic controller, it can be interpreted as the state of the system. This state will probably be associated with an active output signal responsible for a specific working mode of an actuator (servomotor closes the door). This interpretation of a process also involves the change

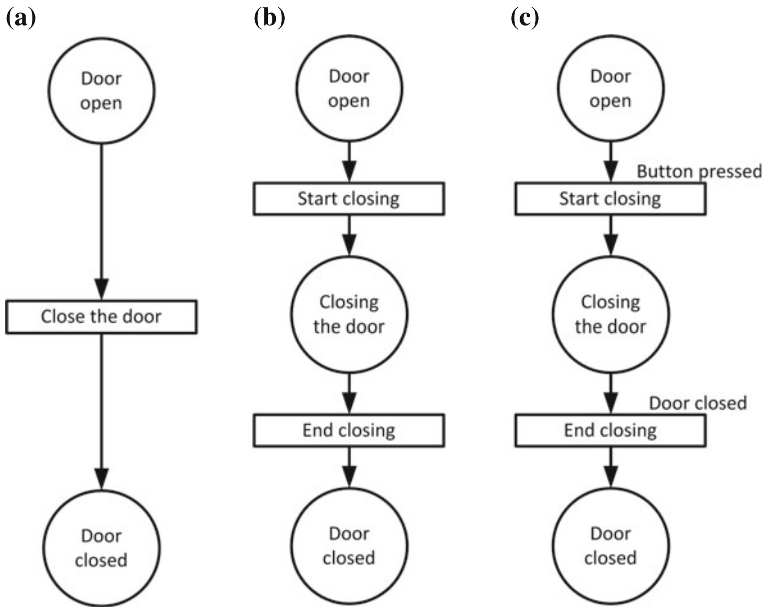


Fig. 4 Possible interpretations of door closing process—dynamic (**a**), state-oriented (**b**) and with condition for starting and stopping (**c**)

from state *Door open* into state *Door closed*. However, between the two listed states in the diagram there will be a state *Closing the door* (Fig. 4b), which corresponds to the action *Close the door*.

After the analysis of the diagram showing subsequent possible states and transitions of the system (Fig. 4b), it can be concluded that there is at least one additional opportunity to transform actions of a UML activity diagram. Figure 4 shows that between successive states there exist transitions *Start closing* and *End closing*. This is the beginning and end of the execution of an isolated process. It can be assumed that these transitions are also part of a particular action.

In addition, it is possible to extend the method by taking into account the specific conditions and situations where the action begins and ends. Considering the arithmetic operations performed by the software, such as adding, the action corresponding to the addition of two numbers starts if these numbers are available, and the end of the action is the receipt of the result (Fig. 5a).

In control systems, however, the start and the end of an action are usually controlled by the input signals to the controller received from the environment or from the controlled object (Fig. 5b). Hence, it must be assumed that the appropriate signal starts the action execution and is a prerequisite for entrance into the action. Start signal can also be considered as a system event triggering the action execution, which is in this case a reaction to the event. In case of automatic door closing it may be *Button pressed*, which appears as an active input signal of a logic controller.

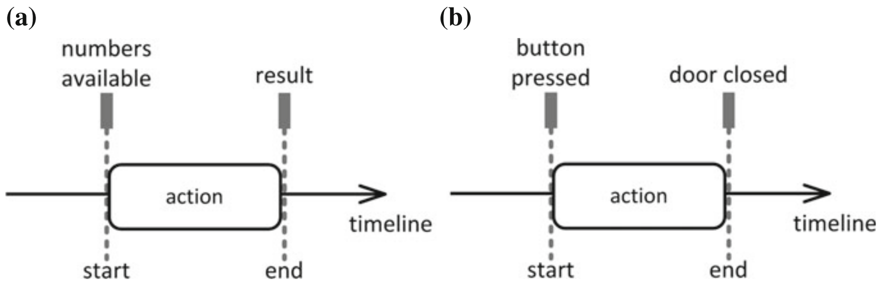


Fig. 5 Actions with starting and ending conditions for software (**a**) and hardware (**b**) systems

Completion of action *Close the door* also depends on the input signal. In the considered example, it can be an active input connected to the limit switch, signaling the complete closure of the door (condition *Door closed*). It is then necessary to deactivate the output signal responsible for door closing, which means the termination of the action (in behavioral specification). Hence, the condition *Door closed* is considered as the output condition of the action, which should be identified with expected response of the controlled object (status) for an operation (action) performed by the logic controller. The above example is illustrated in Fig. 4c in the form of appropriate conditions assigned to transitions.

In a dynamic interpretation of an action, it is possible to assign the *TRUE* value to the logic controller output signal in one action, and then in another part of the diagram to assign the *FALSE* value to it, as it is shown in Fig. 6a. The solution has a big advantage since the particular output signal appears only in two actions. In all elements between these two actions, the output signals hold their values. However, a special attention should be paid to ensure that the output signal, when switched on, will eventually be switched off and that the action responsible for it is reachable.

In contrast to the dynamic interpretation, when considering the state-oriented approach, the particular output signal has to appear in all action and activity nodes when it is supposed to maintain the *TRUE* value, as it is shown in Fig. 6b. In particular, the output signal activity has to be taken into account also by complex activities and one should remember to include the signal in all internal actions of activities.

4 Summary

Logic controller specification can be prepared by means of various techniques [3] and then implemented e.g. in the structures of FPGA type [11, 12]. The chapter is focused on the activity diagrams of UML language (version 2.x) and presents various interpretations of actions in logic controller design.

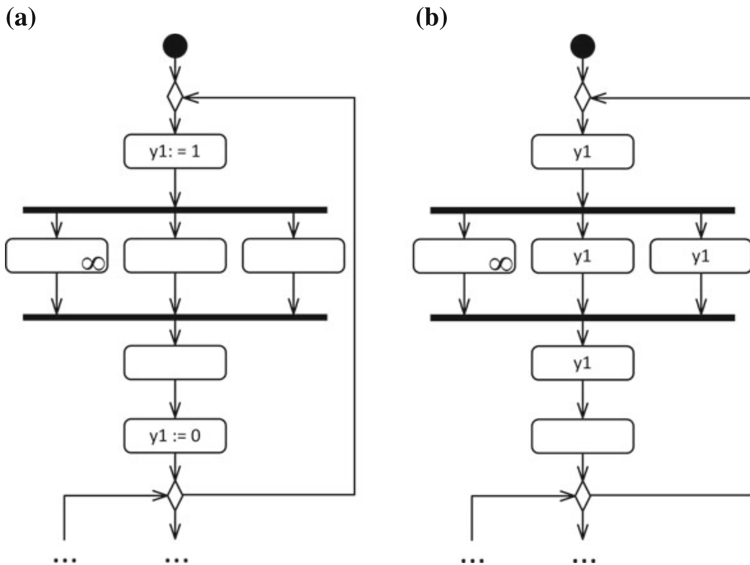


Fig. 6 Logic controller output signals in dynamic (**a**) and state oriented (**b**) approaches

UML has many advantages in comparison to other specification techniques. Most important are here the user-friendly, simple and clear diagrams showing technical aspects in an easy way. The interpretation of activity diagram actions depends on the view on the system and influences the details of behavioral specification.

UML activity diagrams can be transformed into another formal specification model, namely the control Petri nets, basing on the described interpretations, as shown in [6, 7]. The transformation is bidirectional and can be used for a simultaneous work with both techniques or in order to change the specification technique used in the project. It is also possible to formally verify the UML activity diagrams using the model checking technique, analogously to formal verification of other specification techniques using a rule-based logical model [4, 5], as it is shown in another chapter of this book.

References

1. Doligalski, M. (2012). *Behavioral specification diversification of reconfigurable logic controllers* (Vol. 20). Lecture Notes in Control and Computer Science, Zielona Góra: University of Zielona Góra Press.
2. Gajski, D. D., Abdi, S., Gerstlauer, A., & Schirner, G. (2009). *Embedded system design: modeling, synthesis, verification*. Berlin: Springer.
3. Gomes, L., Barros, J. P., & Costa, A. (2006). Modeling formalisms for embedded system design. In R. Zurawski (Ed.), *Embedded systems handbook*. New York: Taylor and Francis Group.

4. Grobelna, I. (2011). Formal verification of embedded logic controller specification with computer deduction in temporal logic. *Przeglad Elektrotechniczny*, *87*(12a), 47–50.
5. Grobelna, I. (2013). *Formal verification of logic controller specification by means of model checking* (Vol. 24). Lecture Notes in Control and Computer Science, Zielona Góra: University of Zielona Góra Press.
6. Grobelny, M., Grobelna, I., & Adamski, M. (2012). Hardware behavioural modelling, verification and synthesis with UML 2.x activity diagrams. In *Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems—PDeS 2012* (pp. 109–114). Brno.
7. Grobelny, M., & Pieczynski, A. (2013). Exception handling in logic controller design by means of UML activity diagrams and control interpreted Petri nets. *Przeglad Elektrotechniczny*, *89*(5), 314–317.
8. OMG. *OMG Unified Modeling LanguageTM (OMG UML) Superstructure ver. 2.4.1*. Object Management Group (2011)
9. Pender, T. (2003). *UML Bible*. Indianapolis: Wiley Publishing Inc.
10. Schattkowsky, T. (2005). UML 2.0—Overview and perspectives in SoC design. In *Proceedings of the conference on design, automation and test in Europe* (Vol. 2, pp. 832–833). Washington: IEEE Computer Society.
11. Tkacz, J., & Adamski, M. (2013). Structured mapping of petri net states and events for FPGA implementations. *International Journal of Electronics and Telecommunications*, *59*(4), 331–339.
12. Wisniewski, R., Barkalov, A., Titarenko, L., & Halang, W. A. (2011). Design of microprogrammed controllers to be implemented in FPGAs. *International Journal of Applied Mathematics and Computer Science*, *21*, 401–412.

Model Checking of UML Activity Diagrams Using a Rule-Based Logical Model

Iwona Grobelna, Michał Grobelny and Marian Adamski

Abstract UML activity diagrams can be used as semi-formal specification of logic controller behavior. On the other hand, formal methods applied at any stage of system development allow increasing in the quality of final products. In the chapter use of the model checking technique to validate the specification against some specified requirements is described. The specification is initially expressed by means of UML activity diagrams and then is transformed to a rule-based logical model suitable both for verification purposes and for logical synthesis for FPGA devices.

Keywords Activity diagrams · Formal methods · Logic controller · Model checking · Specification · Verification · UML

1 Introduction

A logic controller specification can be formally described using various techniques, as for example control interpreted Petri nets [4] or diagrams of the Unified Modeling Language (UML) [14], especially UML activity diagrams or state machines. Each technique demonstrates both some advantages and disadvantages. In the chapter, the UML activity diagrams (in version 2.x) are used to describe formally the behavior of a designed logic controller. The UML itself is a user-friendly, established technique

I. Grobelna (✉)
Institute of Electrical Engineering,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: i.grobelna@iee.uz.zgora.pl

M. Grobelny
Department of Media and Information Technologies,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: m.grobelny@kmti.uz.zgora.pl

M. Adamski
Institute of Metrology, Electronics and Computer Science,
University of Zielona Góra, ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: m.adamski@imei.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_12

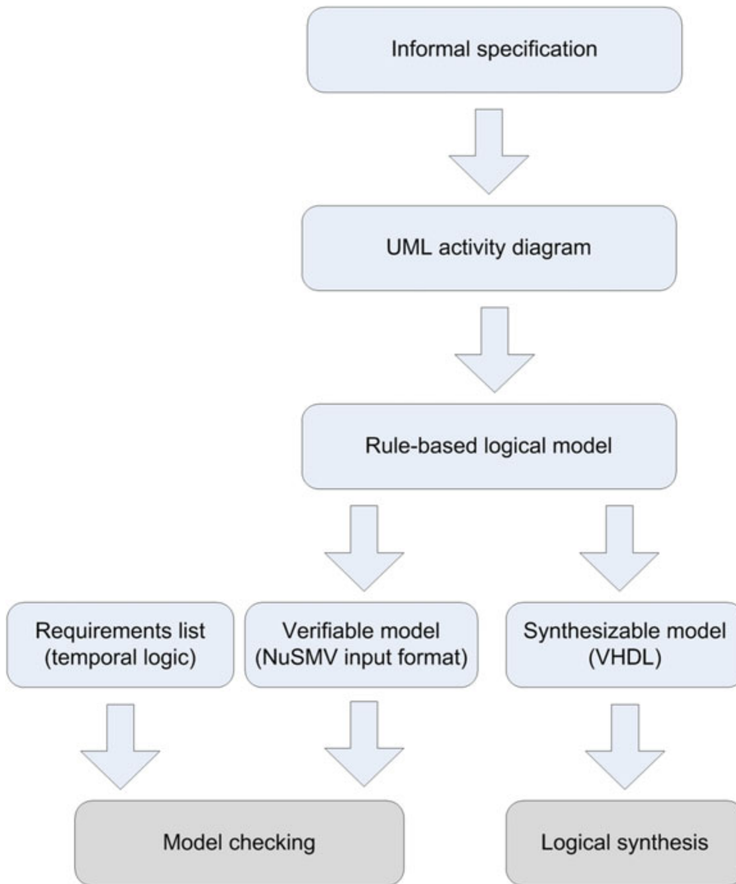


Fig. 1 Usage of a rule-based logical model for model checking of UML activity diagrams

for modelling of software. Commonly, it is also used in a logic controller design process [5, 9, 10, 13].

Former researches have shown that it is possible to specify behavior of the embedded systems using the UML activity diagrams [9, 10]. Unfortunately, the UML diagrams are still not well supported by formal techniques of analysis and verification. On the other hand, control interpreted Petri nets [4], as a mathematical apparatus, have a wide-range support of techniques and mechanisms which can improve the quality of specifications. It is possible to transform a UML activity diagram into a control Petri net [9] and then use the benefits of the second specification technique.

In the proposed approach a rule-based logical model [7, 8] is used as an intermediate format which is suitable both for formal verification [12] using the model checking technique [3, 6] and for logical synthesis (see Fig. 1). It includes rules describing the behavior of a logic controller, and has been primarily based on the

control interpreted Petri nets. The properties of the designed system to be verified are expressed with the temporal logic formulas [11]. Model checking is performed using the *NuSMV* model checker. The rule-based logical model is also a basis for VHDL code, which can be easily implemented in the reconfigurable structures of FPGA type [1]. In this way also a UML activity diagram describing a logic controller can be transformed into a rule-based logical model. It is then possible to verify it using a model checking technique, and next to synthesize it. A big advantage of the proposed solution is that the logic controller can be easily specified in a user-friendly form and at the same time the so-prepared specification can be formally verified and synthesized. So, after formal verification the quality of the final product rises significantly and the implemented device is consistent with its primary specification.

The chapter is structured as follows. Section 2 introduces a rule-based logical model suitable both for formal verification using model checking technique and for logical synthesis. Section 3 shows how a UML activity diagram can be transformed into a rule-based logical model. Section 4 focuses on the model checking of a rule-based logical model, and so of a UML activity diagram. Finally, Sect. 5 summarizes the chapter.

2 A Rule-Based Logical Model

A method for formal verification of logic controller specification expressed as a rule-based logical model has been established [7, 8]. It allows to verify the specification against behavioral properties defined with temporal logic formulas. The rule-based logical model has to be prepared basing either on informal or formal specification of the control process, e.g. control interpreted Petri nets or UML activity diagrams. The proposed rule-based logical model is suitable for model checking as well as for logical synthesis. So, as the result, the implemented solution is fully consistent with the already verified specification.

2.1 Elements of a Rule-Based Logical Model

A rule-based logical model consists of five different sections corresponding to particular elements of specification and presenting its various aspects, described in detail as follows.

Variables definition Variables occurring in the specification of a logic controller: input and output signals as well as places (local states) of the control process. The presence of *places* is connected with the primary usage of a logical model, namely control interpreted Petri nets. However, this section is also well suited to define basic state elements of other specification techniques, in particular actions of UML activity diagrams. It starts with the keyword *VARIABLES*.

Example in Listing 1 shows variables definition for a logic controller specification involving six places, three input and three output signals.

Listing 1 Variables definition in a rule-based logical model

```
1 VARIABLES
2    places: p1, p2, p3, p4, p5, p6
3    inputs: i1, i2, i3
4    outputs: o1, o2, o3
```

Initial values of variables A predefined initial value is assigned to each variable. Then, the value can change according to the rules defined in the next section. It starts with the keyword *INITIALLY*. The exclamation mark is used to express the *FALSE* value, otherwise the variable takes the *TRUE* value.

Example in Listing 2 shows the assignment of initial values to the variables defined in the previous section.

Listing 2 Initial values of variables in a rule-based logical model

```
1 INITIALLY
2    p1; !p2; !p3; !p4; !p5; !p6
3    !i1; !i2; !i3
4    !o1; !o2; !o3
```

Transitions definition In this section, the rules which change the values of variables are defined. The rules correspond to transitions in control interpreted Petri nets or to flows in UML activity diagrams. The section starts with the keyword *TRANSITIONS*. Each rule is given an etiquette to simplify the definition process for the user. A rule itself consists of two parts. The one before the arrow is a precondition, the one after the arrow is a postcondition preceded by the temporal logic operator *X* stating that the change will happen in the next state of the system. Here also the exclamation mark is used to express the negation of a variable.

Example in Listing 3 shows some rules, e.g. the first rule is executed when both variables *p1* and *i1* are *TRUE* and then *p1* changes its value into *FALSE* and at the same time two other variables *p2* and *p3* become *TRUE*.

Listing 3 Transitions in a rule-based logical model

```
1 TRANSITIONS
2    t1: p1 & i1 -> X (!p1 & p2 & p3);
3    t2: p2 & i2 -> X (!p2 & p4);
4    ...
```

Input signal changes The section defines expected changes of the input signal values. It is used only for verification purposes in order to eliminate the so-called state explosion problem [3]. In the real world, input signal changes are caused by the environment. Here values changes are expected to happen in correspondence to particular places (left side of the arrow). Then (right side of the arrow), the appropriate listed input signal may become either active (*TRUE* value) or inactive (*FALSE* value indicated by the exclamation mark). The section starts with the keyword *INPUTS*.

Example in Listing 4 shows some expected changes of input signals, e.g. if the first place (*p1*) is active, then we expect the *i1* input signal to change its value.

Listing 4 Input signal changes in a rule-based logical model

```
1  INPUTS
2     p1 -> !i1 | i1;
3     p2 -> !i2 | i2;
4     ...
```

Output signal changes The section defines expected changes of output signal values. It is used for verification purposes as well as for logic synthesis. Analogously to input signal changes in the rule-based logical model, changes are expected to happen in correspondence to particular places (left side of an arrow). Then (right side of an arrow), the appropriate listed output signal(s) are active. It starts with the keyword *OUTPUTS*.

Example in Listing 5 shows some expected changes of output signals, e.g. if the first place (*p1*) is active, then the *o1* output signal is active.

Listing 5 Output signal changes in a rule-based logical model

```
1  OUTPUTS
2     p1 -> o1;
3     p3 -> o2;
4     ...
```

2.2 Verifiable Model Basing on a Rule-Based Logical Model

A rule-based logical model can be automatically transformed into a verifiable model using the implemented *m2vs* tool according to the following rules.

Rule 1
Each place is a variable of Boolean type.
Rule 2
Each input signal is a variable of Boolean type.
Rule 3
Each output signal is a variable of Boolean type.
Rule 4
Defined variables take some initial values. Each variable takes any of two values *TRUE* or *FALSE*.

Rule 5
Each place changes according to the rules defined in the transitions; conditions of changes between places occur in pairs (groups), in the previous place(s) and in the next place(s).
Rule 6
Each input signal changes randomly, but can take the expected values (connected in this case with actions of a UML activity diagram) or change adequately to the situation.
Rule 7
Output signals changes are defined in corresponding places (in this case actions of activity diagram).

After applying these rules, the *m2vs* software transforms the rule-based logical model into a verifiable model in the *NuSMV* model checker format. Analogously to the rule-based logical model, it starts with variables definition and their initial values. Then, each variable is assigned a value in the next state. Additionally, by input signals value changes one of possible values *FALSE* or *TRUE* is randomly chosen for the next state if a particular place or action is active. Otherwise it maintains the *FALSE* value (in order to eliminate the state explosion). Next, the *NuSMV* tool is used to verify formally the specification using the model checking technique. The process of formal verification itself is described in Sect. 4.

2.3 Synthesizable Model Basing on a Rule-Based Logical Model

A rule-based logical model can be also transformed into a synthesizable model in VHDL language and then simulated and synthesized. It should be noted, that both verifiable and synthesizable models are consistent with each other. Hence, the obtained physical implementation is consistent with the primary, already verified, specification. The transformation into VHDL code is performed automatically using the implemented *m2vs* tool. The prepared model can be then easily simulated (using e.g. *Active-HDL* environment) and synthesized (using e.g. *XILINX Plan Ahead* environment). Synthesis is performed in a form of rapid prototyping [1, 2] where optimization aspects are out of scope. Its main goal is to check whether the designed system operates at all and some redundant hardware elements may be used.

3 UML Activity Diagram as a Rule-Based Logical Model

A simplified UML activity diagram can be defined as a seven-tuple $AD = \{A, T, G, F, S, E, Z\}$ with:

- A being a set of actions/activities;
- T being a set of transitions (e.g. fork and merge nodes);
- G being a set of guard conditions corresponding to transitions (input signals);
- F being a set of flow relation between the activities and transitions;
- S being a set containing an initial node;
- E being a set containing a final node;
- Z being a set of output signals.

An activity diagram describing logic controller behavior can be represented as a rule-based logical model in two steps described as follows.

3.1 Step One—Labelling

Each action $a \in A$ is labelled with an etiquette aX, where X stands for the number of action. Moreover, the initial node S is labelled as *aStart* and the final node E as *aEnd*. Each transition $t \in T$ is labelled with an etiquette tX, where X stands for the number of transition. The labelling is illustrated in Fig. 2.

3.2 Step Two—Notation

Basing on a labelled UML activity diagram, a rule-based logical model is built according to four rules defined as follows. Despite the labelling itself, output signals listed inside actions and input signals occurring at transitions or flows are taken into account.

Rule 1 for actions (*A*) together with initial (*S*) and final (*E*) node
The set of actions, with the initial and final node, are mapped in a section for variables (as *places*, keyword *VARIABLES*). The initial state of the system is reflected in a section for initial values (keyword *INITIALLY*). Usually, only the initial node is active. Then, actions, initial and final nodes appear also in other sections of the logical model.

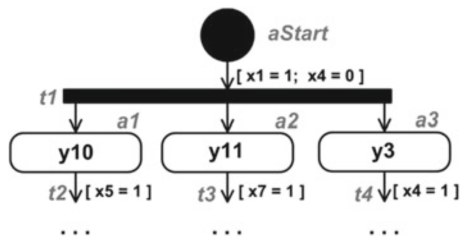


Fig. 2 A sample UML activity diagram with labelling

Rule 2 for transitions (T) and flow relations (F)
The set of transitions and the set of flow relations between the actions and transitions are used for rules definition. Flows between actions are treated as transitions, the same as fork and join nodes. Transitions are mapped in section *TRANSITIONS*, each one in a separate row. The rules take into account actions, the initial and final node, as well as logic controller's input signals which are treated as conditions.
Rule 3 for input signals (G)
The set of input signals is mapped in a section for input definition (as *INPUTS*, keyword *VARIABLES*). Then, input signals obtain certain predefined values, usually none of them is active. Input signals appear also in rules definition. A special section for inputs (keyword *INPUTS*) indicates expected value changes in correspondence to particular actions.
Rule 4 for output signals (Z)
The set of output signals is mapped in a section for outputs definition (as *OUTPUTS*, keyword *VARIABLES*). Then, output signals take some predefined values, usually none of them is active. Output signals do not appear in rule definition. A special section for outputs (keyword *OUTPUTS*) indicates when the defined output signals are active.

The usage of the rules is illustrated in Listing 6 where a part of the rule-based logical model is shown for a sample part of UML activity diagram from Fig. 2. And so:

- basic keywords include lines 1, 5, 9, 13 and 17;
- adapting rule 1 results in lines 2, 6, 10–12, 14–16 and 18–20;
- adapting rule 2 results in lines 10–12;
- adapting rule 3 results in lines 3, 7, 10–12 and 14–16;
- adapting rule 4 results in lines 4, 8 and 18–20.

Listing 6 Rule-based logical model of a sample UML activity diagram

```
1  VARIABLES
2  places: aStart, a1, a2, a3, ...
3  inputs: x1, x2, x3, x4, x5, x6, x7, ...
4  outputs: y1, y2, y3, ..., y10, y11, ...
5  INITIALLY
6  aStart; !a1; !a2; !a3; ...
7  !x1; !x2; !x3; !x4; !x5; !x6; !x7; ...
8  !y1; !y2; !y3; ...; !y10; !y11; ...
9  TRANSITIONS
10 t1: aStart & x1 & !x4 -> X (!aStart & a1 & a2 & a3);
11 t2: a1 & x5 -> X (!a1 & ...)
12 ...
13 INPUTS
14 aStart -> (!x1 | x1) & (!x4 | x4);
15 a1 -> !x5 | x5;
16 ...
```

```
17  OUTPUTS
18  a1 -> y10;
19  a2 -> y11;
20  ...
```

4 Model Checking of a Rule-Based Logical Model

Model checking process gives an answer whether the defined system model satisfies the list of requirements specified as temporal logic formulas. The defined system model (a verifiable model) is already generated. To verify formally the model description, requirements list is also needed (Fig. 3). The list includes properties which are supposed to be fulfilled in the designed embedded system, they include structural as well as behavioral properties. Requirements are defined using either Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) formulas, as e.g. the CTL formulas:

$$AG(x5 \rightarrow EFy2 \wedge y8) \tag{1}$$

$$AG\neg(y9 \wedge y10) \tag{2}$$

$$EFaEnd \tag{3}$$

The definition of formulas is usually based on an informal specification of control process, and it is most desired if they are delivered either by a customer or by any other team in the company. The most frequently verified properties regard safety (*something bad will never happen*), e.g. formula (2), and liveness (*something good will eventually happen*), e.g. formula (3). In the verification process mostly behavioral requirements are taken into account which include activities of input and output

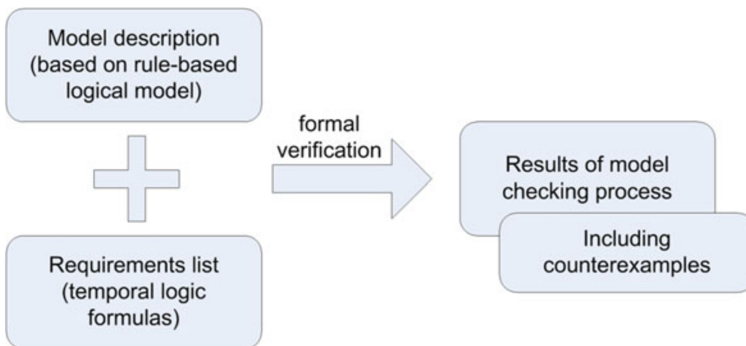


Fig. 3 Model checking of a rule-based logical model

signals. And so, formula (1) states that always when input signal *x5* is active, then finally both output signals *y2* and *y8* can become active. Formula (2) states that it should never be the case that both output signals *y9* and *y10* are active at the same time. Formula (3) checks whether the final node is reachable at all.

When both parts needed for formal verification are prepared (Fig. 3), the *NuSMV* model checker compares the model description with the delivered requirements list and gives an answer whether the properties are satisfied in the designed system. If this is not the case, appropriate counterexamples are generated which allow finding the error source in the model description. Then either the specification or the particular unsatisfied requirement has been incorrectly formulated.

5 Summary

The chapter presents a novel approach to formal verification of UML activity diagrams (in version 2.x) describing a logic controller behavior. An original rule-based logical model is used to ensure the consistency between a verifiable model and a synthesizable model. Formal verification is performed using the model checking technique and the *NuSMV* tool. Requirements to be verified are expressed as the temporal logic formulas. The transformation of a rule-based logical model into a verifiable model and a synthesizable model is done automatically using the implemented *m2vs* tool.

The results of the research show that it is possible to use the commonly-known and user-friendly UML language in logic controller design, focusing in particular on activity diagrams [10]. Using the proposed rule-based logical model [7, 8] it is possible to verify the specification formally using the model checking technique as well as to synthesize it in the reconfigurable structures of FPGA-type. Finally, the implemented solution is consistent with the previously formally verified logic controller specification expressed by means of UML activity diagrams.

References

1. Ahrends, S. (2008). *Neue Ansätze für effizientes Rapid Prototyping von Embedded Systemen. Embedded Computing Conference*
2. Andreu, D., Souquet, G., & Gil, T. (2008). Petri net based rapid prototyping of digital complex system. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI* (pp. 405–410). Washington: IEEE Computer Society.
3. Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge: The MIT Press.
4. David, R., & Alla, H. (2010). *Discrete, continuous, and hybrid Petri nets* (2nd ed.). Berlin: Springer.
5. Doligalski, M., & Adamski, M (2013). UML state machine implementation in FPGA devices by means of dual model and Verilog. In *11th IEEE International Conference on Industrial Informatics - INDIN* (pp. 177–184). Bochum, Germany. ISBN: 978-1-4799-0751-9

6. Emerson, E. A. (2008). The beginning of model checking: A personal perspective. In O. Grumberg & H. Veith (Eds.), *25 years of model checking* (Vol. 5000, pp. 27–45). Lecture notes in computer science. Berlin: Springer.
7. Grobelna, I. (2011). Formal verification of embedded logic controller specification with computer deduction in temporal logic. *Przeglad Elektrotechniczny*, *87*(12a), 47–50.
8. Grobelna, I. (2013). *Formal verification of logic controller specification by means of model checking* (Vol. 24). Lecture notes in control and computer science. Zielona Góra: University of Zielona Góra Press.
9. Grobelna, I., Grobelny, M., & Adamski, M. (2010). Petri nets and activity diagrams in logic controller specification - transformation and verification. In A. Napieralski (Ed.), *17th International Conference on Mixed Design of Integrated Circuits and Systems - MIXDES* (pp. 607–612). Wroclaw.
10. Grobelny, M., Grobelna, I., Adamski, M. (2012). Hardware behavioural modelling, verification and synthesis with UML 2.x activity diagrams. In *Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems - PDeS 2012* (pp. 109–114). Brno.
11. Huth, M., & Ryan, M. (2004). *Logic in computer science: Modelling and reasoning about systems*. New York: Cambridge University Press.
12. Kropf, T. (1999). *Introduction to formal hardware verification: Methods and tools for designing correct circuits and systems* (1st ed.). New York: Springer.
13. Łabiak, G., Adamski, M., Tkacz, J., Doligalski, M., & Bukowiec, A. (2011). Role of UML modelling in discrete controller design. In D. Zydek & H. Selvaraj (Eds.), *Proceedings of 21st International Conference on Systems Engineering ICSEng 2011* (pp. 480–481). Las Vegas, USA. University of Nevada, IEEE Computer Society. ISBN: 978-0-7695-4495-3.
14. OMG (2011). *OMG Unified Modeling LanguageTM (OMG UML) Superstructure ver. 2.4.1*. Object Management Group.

UML Support for Statecharts-Based Digital Logic Controller Design in FPGA Technology

Grzegorz Łabiak

Abstract The paper describes usage of UML methodology in digital logic control modeling, which is one of few stages of digital logic controller development life cycle. The digital logic control modeling process is compared with traditional and well known software development methodology. In the comparison the differences are particularly emphasized. The main differences are connected to analyzing process and modeling aims. In case of software development crucial role plays object analysis which is meant to bring creation of data model. In case of digital logic controller design main activity in modeling is behavior analysis which is aimed to specify formally and precisely controller behavior.

Keywords UML · Digital logic controller · Methodology · Conceptual modelling · Behavior analysis · State machine

1 Introduction

Digital logic controller design is a process which traditionally can be performed as a Finite State Machine design. The main drawback of this methodology is that in case of complex behavior number of states of the automaton can grows exponentially, and the complexity of the project grows also. An option for the designer is to use the concurrency and/or hierarchy paradigms, which efficiently reduces complexity of the design. Computational complexity is not the only problem in logic controller design. Even more complicated is preliminary modeling, which has to be performed before a design is formalised. This stage, called conceptual modeling, is an activity where an ordering client formulates his wishes and functions of the system under design and the engineer must give them a material form. The language which supports this conversation, which is both general and detailed, is Unified Modeling

G. Łabiak (✉)
Institute of Electrical Engineering, University of Zielona Góra,
ul. Licealna 9, 65-417 Zielona Góra, Poland
e-mail: G.Labiak@iee.uz.zgora.pl

© Springer International Publishing Switzerland 2016

A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8_13

Language (UML) [4, 20]. From an ordering client point of view UML diagrams are very general and easy to understand, but from engineering point of view they offer enough syntax features to describe the systems in details.

UML diagrams, originated in software engineers environment [4], turn out to be perfectly fitted into phases of discrete system development life cycle. In proposed methodology main paradigm shift is a new role of object-oriented analysis and a new role of state machine diagrams. The main goal of object-oriented analysis in digital logic controller design is to identify the real word objects of the system (elements of the controlled object) and then to express in terms of these objects functionality of the system. The state machine diagrams role is a kind of transition between an imprecise UML model and a formal unambiguous description of the controller behaviour. This transition through author's semantic interpretation, extends UML and provides that the state machine diagrams can be synthesized and implemented in Field Programmable Gate Arrays (FPGA).

2 Digital Design for FPGA

Technology development in the seventies and eighties of the last century has allowed for new approach in manufacture of the digital circuits. Until then digital circuits were produced in technologies which required substantial participation of the device manufacturer, without which obtaining final functionality of the circuit was impossible. Electronic devices were produced and programmed solely in factories. In the mid-80s appeared new technologies which allowed for programming electronic devices outside the factory. Among them, in 1985 Xilinx corporation introduced first device in new technology called Field Programmable Gate Arrays (FPGA).

Architecture of FPGAs consists of an array of programmable logic blocks (Fig. 1) interconnected through special programmable matrices. Every logic block can be configured as a combinational logic with a flip-flop. Contemporary FPGA devices have logic blocks which allow to programm logic function up to 6 variables implemented as Lookup Table. Logic blocks through interconnections can be freely connected. This technology gives designers a new potential, because volume of available resources in one device is of the order of tens of thousands of logic blocks.

The new technology created new possibilities for designers [8]. They had at their disposal huge amount of hardware resources, which were accessible from the shelf. With time, according Moore's law the number of transistors in a integrated circuit doubles approximately every two years, the technology gap has arisen between accessible hardware resources and the methods of digital circuit design. Designer were not able to use all accessible hardware resources offered by producers in a single programmable device. This situation provoked scientists to developed new design methods which would be able to manage more complicated and functional projects demanding more gates and flip-flops.

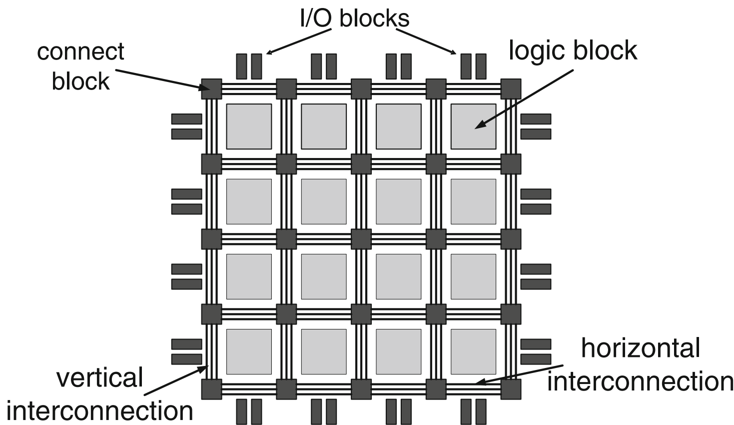


Fig. 1 Architecture of FPGA

One of the most important result of the research in the filed of digital design methodology is a development of Hardware Description Languages (HDLs). HDLs allow designers to describe a digital electronic circuit formally at high level independently of final technology [22]. Nowadays HDLs are not only formal description of the design, but also serve as a standardized vehicle of the information exchange between Computer Aided Design software (CADs) [7]. The most prominent languages are VHDL and Verilog.

Much of the recent scientific effort in digital design methodology is put into a conceptual modeling. The conceptual modeling is a first material activity done by a designer after conceptualization which takes place in human mind. The objective of conceptual modeling is to express human intensions and to give them material form eg. diagrams. In case of a design of a digital circuits the most often are modelled functionality to the device, its behavior or its structure. The technology applied in this field is Unified Modeling Language (UML) with their dedicated special profiles [9, 13] like MARTE profile [19] or UML Profile for System on Chip (SoC) [6, 18].

Figure 2 presents design flow for the controllers implemented in FPGA technology. Every design typically starts with vague ideas about design functionality and its structure. At this beginning stage project manager and an ordering client are talking in general terms, using very imprecise vocabulary describing their concepts and ideas. Moreover, technical details can not be known at this stage and must be gradually specified. Usually the client is not an engineer and can not be precise, hence some technology and financially depended technical details must be defined by engineers and submitted to the client for acceptance. This feasibility study must be documented in the language which is on the one hand very easy for comprehension for non-engineers, on the other hand is precise and detailed enough for engineers to realize what was ordered by the client. The langue which has both two paradoxically opposing features is Unified Modelling Language (UML) [17, 21]. UML is a

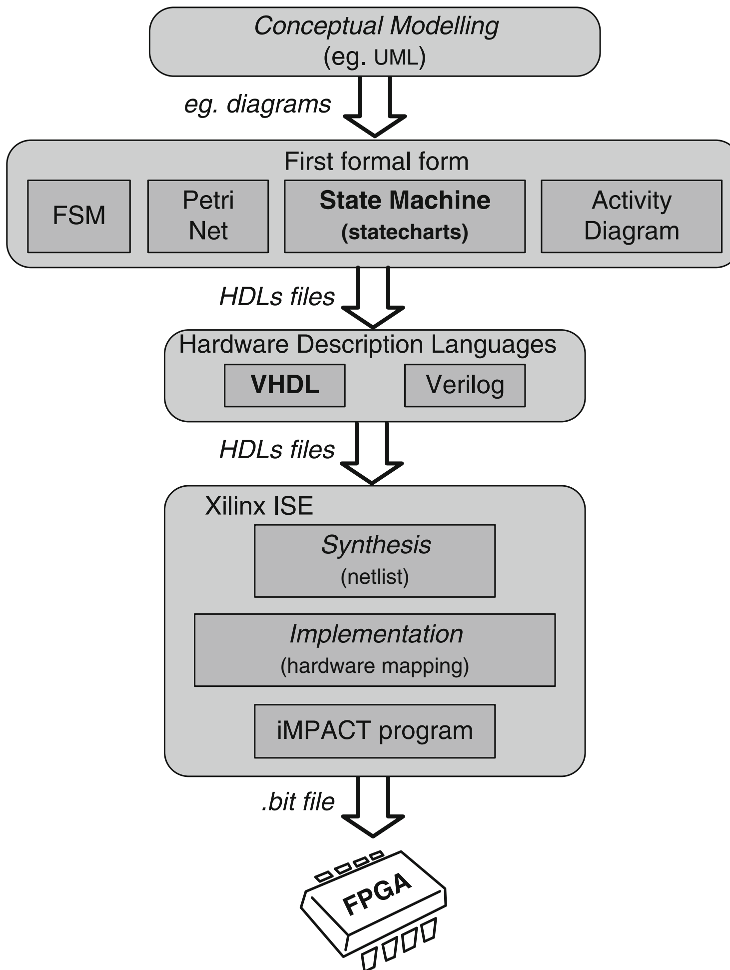


Fig. 2 Design flow of a controller in FPGA technology

technology developed by software engineers and a class diagram is a link between informally specified project (in terms of UML called a model) and programmers. In case of hardware design and particularly in digital logic controller case, this link can be made by state machine diagram (statecharts) or activity diagram. Dedicated Computer Aided Design (CAD) software transforms description of a controller behavior into HDL (eg. VHDL [15, 16] or Verilog [1, 2]) and the project in this form is an entry to commercial CAD system supported by the producer of FPGA devices. In case of Xilinx devices the software is Xilinx ISE (Integrated Synthesis Environment). Xilinx ISE after compilation acts by stages of a synthesis and an implementation which are heavily technology dependant ones. The synthesis process maps design logic structures to devices primitives (eg. CLBs or flip-flops) and the implementation process

places the netlist generated at the synthesis stage onto the target device. Finally, iMPACT program transfers a *.bit* file (generated at the implementation stage) into the devices installed on a printed circuit board.

3 Digital Logic Controllers

Digital logic controllers belongs to the broader class of systems called computer reactive system. A computer reactive system is a system that changes its behavior (actions, outputs, transitions) in response to stimuli coming from outside world or from internal events. Reactive systems are event driven systems continuously having to react to external/internal stimuli. Their response to the stimuli is immediate for an observer, unlike transformational systems which wait for readiness of data on its inputs, and than after some time spent on processing data, the system signals readiness of data on its outputs. FPGA devices are very efficient technology to implement computer reactive systems executing logic control. The most often these systems are designed according to two popular models: controller with an object and control unit with datapath [10].

A control system is the system of linked controlled object and control unit called controller (Fig. 3). The controller has programmed functionality and affects on working of controlled object. The controlled object is a kind of dynamic system whose desired behavior can be stimulated by the controller by means of control signals. An example of control object is an industrial process (eg. assembly line) or chemical reactor (eg. producing penicillin). The controller is a device, which receives information (state signals) about a state of the controlled object. These signals tell the controller about changes in the object, and on this information are based decisions taken by the controller which affects desired course of the controlled process. Additionally, a control process can run with a participation of a man (an operator), who by means of operator's signals can also affect controlled process (eg. in the event of a breakdown) as well as is informed about the course of the process.

The control in above mentioned system is as a closed loop control, because by means of state signals the control object affects the controller. The controller initiates process in the system (is a cause) and the behavior of the controlled object is a result (an effect) of the controller's interaction. The state signals of the object serve as a feedback and provide classical affecting of an effect on its cause.

Next model of a control system (Fig. 4) is called control unit with datapath. In this scheme input data processing takes place in the datapath, which is made of functional blocks such as arithmetic logic unit, multiplexers or registers interconnected with buses. Subsequent stages of data processing are governed by control unit, which based on status signal from datapath takes the decision about further processing and by means of control signals transmit this information to the datapath. This control model is well suited to the systems which process huge amount of data such as video or sound streams. Then, the algorithm processing data is implemented in the control unit and the processing it is contained in the datapath. The algorithm appears to the

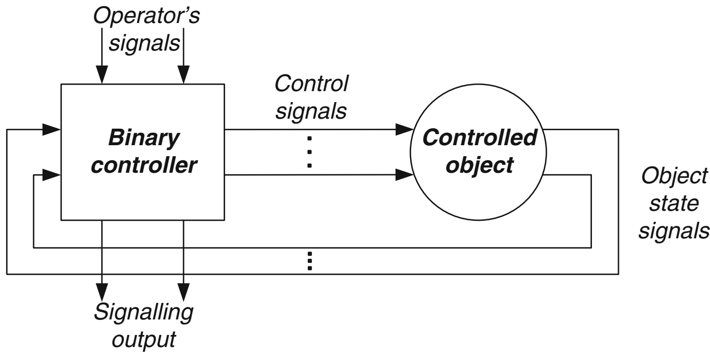


Fig. 3 Discrete control system

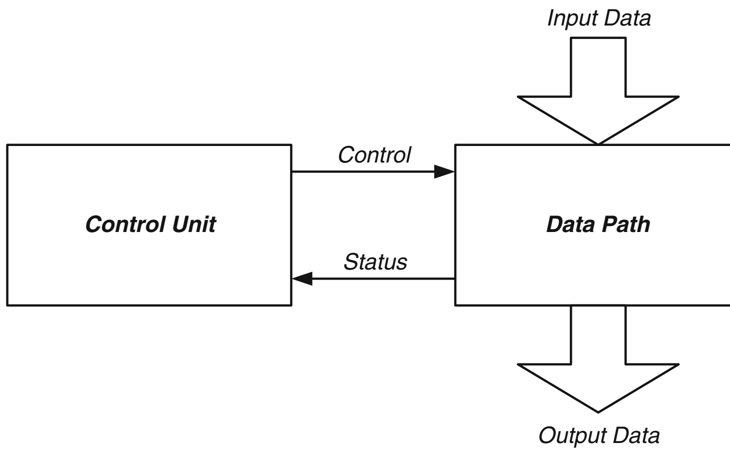


Fig. 4 Control unit with datapath

designer as a traditional control problem and can be designed according classical paradigms like FSM, interpreted Petri nets or statechart diagrams. General purpose processors are designed according to this model.

4 UML in Discrete System Development Process

4.1 Controller Design

Digital logic controller design, traditionally, can be carry out in PLC technology or as a digital circuit [5]. The former, as its first formal description of behavior, mostly uses Ladder Diagram (LD) or Sequential Function Chart (SFC). The latter uses very

often following paradigms: Finite State Machine (FSM) [22], Petri nets [3] and state machine (statechart) [12, 16]. The mentioned paradigms result from the division of modeling methodologies taking into account sequentiality, concurrency and hierarchy. In practice, engineers use more specialized and more refined methodologies based on the state oriented FSM, mainly, Concurrent FSM (realized by interpreted Petri nets), Hierarchical Concurrent FSM (HCFSM, realized by statecharts or UML state machines) [10] and special application of UML activity diagrams combining concurrency and hierarchy [11]. Having project done in the first formal form, further designing processes can be fully automated using at most small assistance of a man. Common feature of these notations is that their first formal form of a controller behavior is preceded by conceptual modeling phase.

4.2 Conceptual Modeling

Regardless of applied technology or formal modeling paradigm, at conceptual modeling stage there is a need to talk on the behavior of the controller under design in informal way. Not every member of the group of people involved in the project is a specialist. The group is rather diverse, beginning with a client ordering project end ending with engineers implementing the device. Each of them represents different skills and culture. What is the most important to them to use a language equally understood by them all. Main goal of this stage is to capture main functionality of the system and its basic architecture. Moreover, as systems have become increasingly complex, the role of conceptual modeling has dramatically expanded. A notation which meets these requirements for conceptual modeling is graphical Unified Modeling Language which offers common vocabulary to talk about the design [21].

4.3 UML in Controller Design

The UML is a notation which was created and in the beginning developed by Grady Booch, Ivar Jacobson and James Rumbaugh in the field of software engineering. Its apparent success in the field has provoked wide interested among scientists and led to the development new methodologies applied in the new areas of technology. Since 1997 UML was adopted by the Object Modeling Group (OMG) and OMG is responsible for its standardization.

Current version of UML defines its objectives as “to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes” [20]. These UML objectives coincide with objectives of digital logic controller design and the application of UML in the design process is also similar. Figure 5, from methodology design perspective, presents four main stages

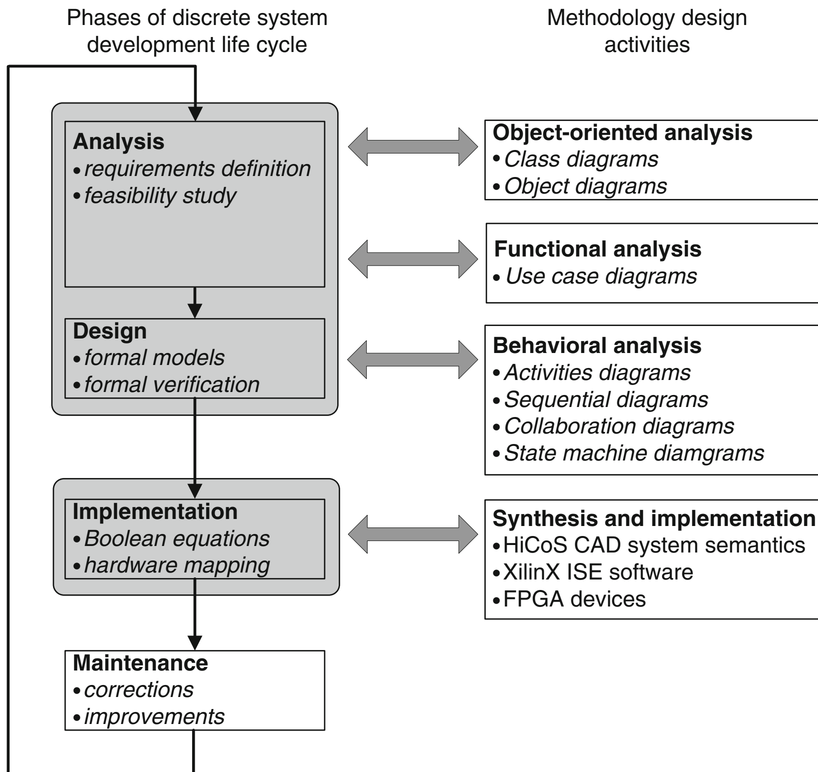


Fig. 5 System development life cycle and design activities

of discrete system development life cycle. The stages presented on shaded area are where UML (in general) and design flow in FPGA (Fig. 2) apply.

Digital logic controller design begins with *analysis* of a control system. Designer must grasp a general idea of workings of a controlled object and a control process which occurs in the object. The goal of the analysis process, called *object-oriented analysis*, is an identification of material object of the real world (i.e. control system), eg. a controller, an operator, a scale, a valve. This preliminary activity is very similar to object-oriented analysis in software engineering, with the difference that designer does not identify abstract relations between the objects like, for example, inheritance. The UML diagrams used at this stage are class diagrams and object diagrams. Next, having identified main objects in the system, the designer defines the roles the objects play in the system. The fundamental goal of such *functional analysis* is to describe what the system performs. The UML diagrams mainly applied at this stage are use case diagrams, which additionally can present relationships between functionalities. The analysis is a right moment to define system's non-functional requirements and to perform feasibility study.

After object and functional analysis stage the designer goes to *design process* and focuses on the details of the system and especially on the process or processes which occur in the system. This stage from the methodology design point of view is called *behavioral analysis* and its purpose is to describe how earlier defined functionalities are executed and which objects take part in them. The UML diagrams useful for behavioral analysis are activity, sequence, collaboration and state machine (statechart) diagrams. The model created in these analyses by definition is not formal and is a framework for the subsequent formalised design process. The main goal of the design process is to produce formal description which is to be an entry form (eg. HDL) to automated CAD system (eg. Xilinx ISE).

The transformation of informal UML model into formal description is a part of an *implementation process* and is out of UML specification. This transformation is an author's proposition of specialized extension of UML for digital controller design [14]. The extension is technology related and is aimed at FPGA devices [5, 16]. In this process author proposes to use state machine (statechart) diagrams as main modeling vehicle. On the one hand UML offers comprehensible graphical notations, on the other the model is informal and must be specific enough to be an entry form for synthesis process. The process of formal specification is conducted according to author's semantics interpretation of state machine diagram and is realized by dedicated academic HiCoS CAD system [15]. Because implementation process of digital logic controller design is not directly supported by the UML and is only author's extension, this process on the Fig. 5 is presented on the separated greyed area. It is noteworthy, that the role played by the state machine diagrams is to link informal UML model with formal requirements of synthesis process. The state machine role of author's methodology is very similar to class diagrams role in software engineering modeling.

5 Example

Let as an example serves simple chemical plant (reactor) whose schematic diagram is presented in Fig. 6. The general objective of the plant is to create new mixture from two components through controlled chemical process. For the designer this plant appears to be discrete control system composed of a controller and controlled object (Fig. 3) with a participation of an operator.

5.1 Object-Oriented Analysis

The first step in the design is a talk of the designer with ordering client. During this, both sides of the talk create own vocabulary and decide what the system is and what the system does. This preliminary talks are informal yet and the goals of these talks are three: (1) identification of the component objects of the system, (2) to set the

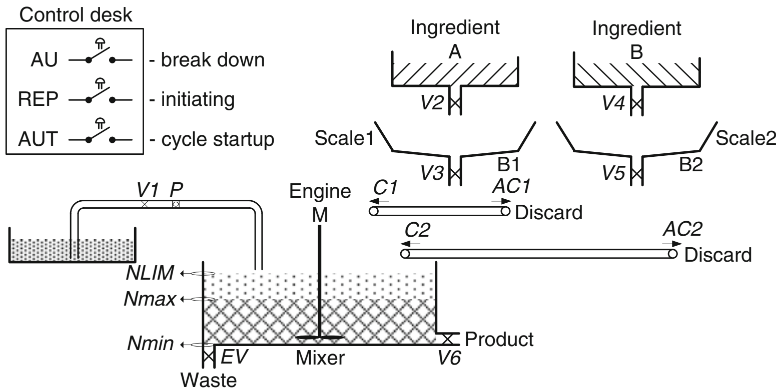


Fig. 6 Schematic diagram of the chemical plant

functions of the objects and (3) to create a verbal description of the process in the controlled object. These three goals in course of the design are successively refined with the use of UML notations.

As far as reactor is concerned (Fig. 6) basic objects of the system are two scales, two conveyor belts, a pump, a mixer and a main container. For complete modeling reasons it is good to set apart the operator object and the controller object as actors.

Useful UML diagrams at this stage of modeling are object diagrams and class diagrams. This analysis is a bit different than object-oriented analysis in software engineering. The main goal of the analysis is to identify and name real objects in the system without creation relationships between them. The names of the object create a vocabulary of the UML model.

5.2 *Functional Analysis*

Having named component objects in the systems, the designer can formulate basic functions of the controlled object and describe them in verbal form. In case of the reactor its working is as follows. Technological cycle consists of three stages: (1) *Initiating*, (2) *Filling* and (3) the *Process*. At the *Initiating* stage the remains of the previous technological process are removed from the conveyor belts (signals *AC1* and *AC2*) and main container is emptied (*EV*). On the stage *Filling* scales 1 and 2 measure out substrates of the chemical process (*V2*, *V4*, *B1*, *B2*). Parallel to weighing the pump (*V1*, *P*) pours water to the main container. After water and substrates are prepared (*Nmax*, *B1*, *B2*) main chemical process (*Process*) starts. For given period of time (*FT1*) the substrates are introduced into the water and blended (*M*), then main container is emptied (*V6*) for time *FT2* and the process is terminated. When the operator presses the button *AUT* the technological process starts again. In case of break-down the operator presses the button *AU*.

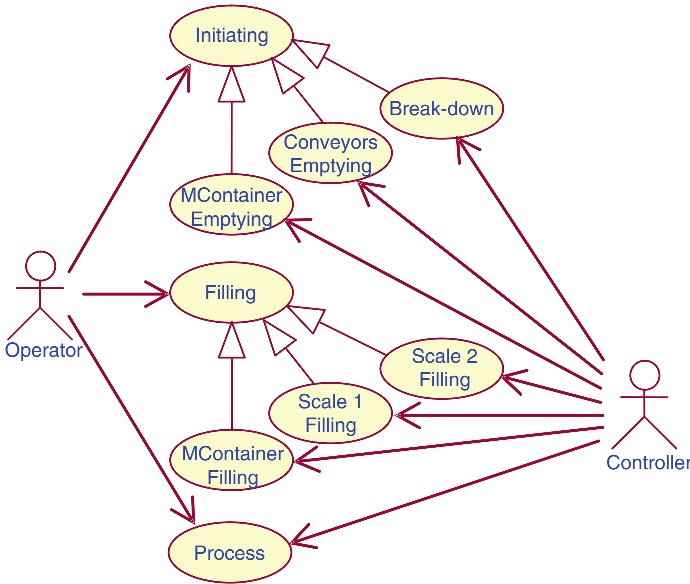


Fig. 7 Use-case-diagram of the chemical plant

The UML diagrams applied at this stage are mainly use case diagrams. Figure 7 presents well developed use case diagram, but for the beginning stage of the design only operator's perspective is essential. It is noteworthy, that the use cases in ovals are not use cases of the system but are use cases of the controlled object.

5.3 Design and Behavioral Analysis

The behavioral analysis is a next design stage supported by UML. In discrete system development life cycle (Fig. 5) this analysis corresponds to a design stage. One of the main goal of a modeling (and also a designing) is to create formal description of the controller behavior, precise enough to be an entry form to CAD systems (eg. ISE Xilinx). In digital logic controller design this goal is different than modeling in software engineering. The software engineer models to build model of data in class diagram (eg. class hierarchy), which is starting point for coders. The controller designer models to create model of behavior, formal enough for synthesis and implementation processes. It seems that UML state machine diagrams are the best diagrams for this purpose. State machines are state oriented, support concurrency and hierarchy and are perfect solution for modeling of behavior of complex control systems.

The behavior analysis can be supported by the following UML diagrams: activities, sequence, collaboration and state machine. These diagrams are assumed to be informal, imprecise, they give general views of the behaviour from different

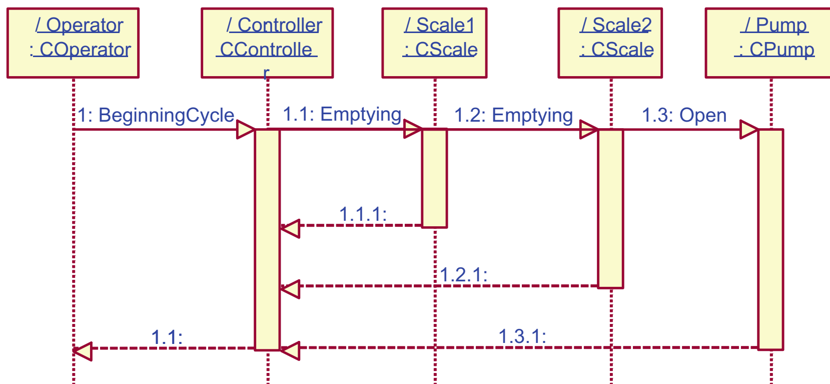


Fig. 8 Sequence diagram for *Filling* use case

perspectives. One of their main goal is to validate client requirements and to give solid basis for formal description. Here, author's proposition is to use UML state machine for this purpose.

After having use case diagram prepared the designer can separately model every use case in details. This can be done by means of sequence diagrams (eg. Fig. 8). Sequence diagram has two dimensions. Horizontally—there are placed objects which take part in modelled functionality (eg. *Filling*) and vertically—there is an axis of time. Arrows represent communications (or procedure callings) among objects. These diagrams present execution a functionality or a task from objects perspective. Figure 8 presents sequence diagram for the use case *Filling*. The object which starts sequence activities is an *Operator*, who by pressing button *AUT* initiates sequence of preparing substrates and water. Other application of the sequence diagrams is in a validation of the controller, namely, the sequence diagrams are ready testing scenario for particular use case.

Next to the sequence diagrams, the activity diagrams also model behavior in details but from different perspective. They concentrate on a system as a whole and show how different elements of the system actively take part in modelled functionality or particular behavior. For example, Fig. 9 presents activity diagram for failure-free work of the reactor from the operator perspective. In this working mode for the operator three objects are most essential: a *Main container* and two *Scales* (*1* and *2*). The three objects in the diagram have assigned its own activity field called swimline. Activities (rectangles with rounded sides) located in the swimline refer to the activity of the object assigned to the swimline. The operator initiates working by pressing *REP* button, then the system executes initiating procedure (activity *Initializing*), next the operator presses *AUT* button. This starts the main technological process—in the diagram compound of two main activities *Filling* and *Process*. The activity *Filling* is presented in details, by its subactivities referring to the objects *MainContainer*, *Scale 1* and *Scale 2*, and the details of *Process* activity are skipped in this diagram.

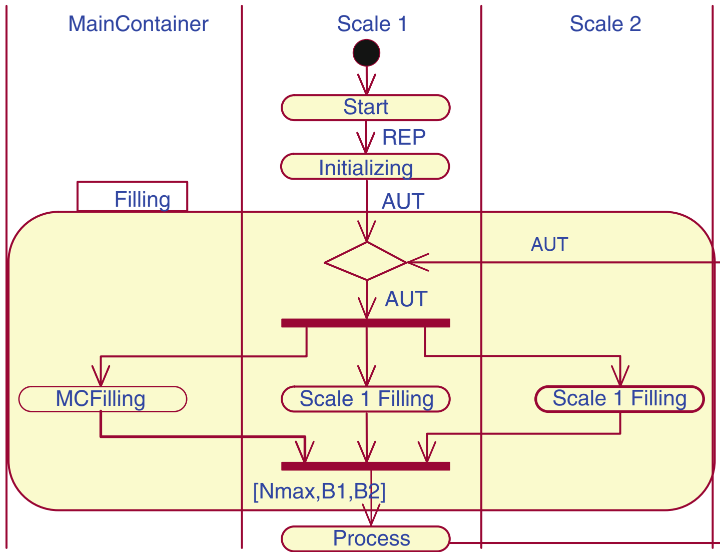


Fig. 9 Activity diagram for failure-free work

5.4 *Implementation*

The implementation phase of a live cycle of a digital logic controller is mainly technology-dependant with little influence of the designer on functionality of the system, if at all. Two main technological processes of this phase are logic synthesis and hardware implementation. Logic synthesis transforms an abstract form of the controller into logic gates (netlist) at Register-transfer level (RTL). Next, hardware implementation makes that netlist obtained in the synthesis process is fitted into target FPGA device. This development phase is not directly supported by UML, but UML provides some language features to extend its applicability. Author's idea of applying UML in digital controller design consists in formulating semantic interpretation of state machine diagrams aimed at implementation in FPGA devices. This UML extension has been implemented is academic CAD system called HiCoS [15]. Figure 10 presents state machine diagram of the controller which precisely and formally defines behavior of the controller. This diagram is unambiguously mapped into textual form *.SSF* (Statechart Specification Format) file, which is an input file for HiCoS system. HiCoS system automatically transforms it into VHDL file which, in turn, is synthesised and implemented by commercial ISE Xilinx software.

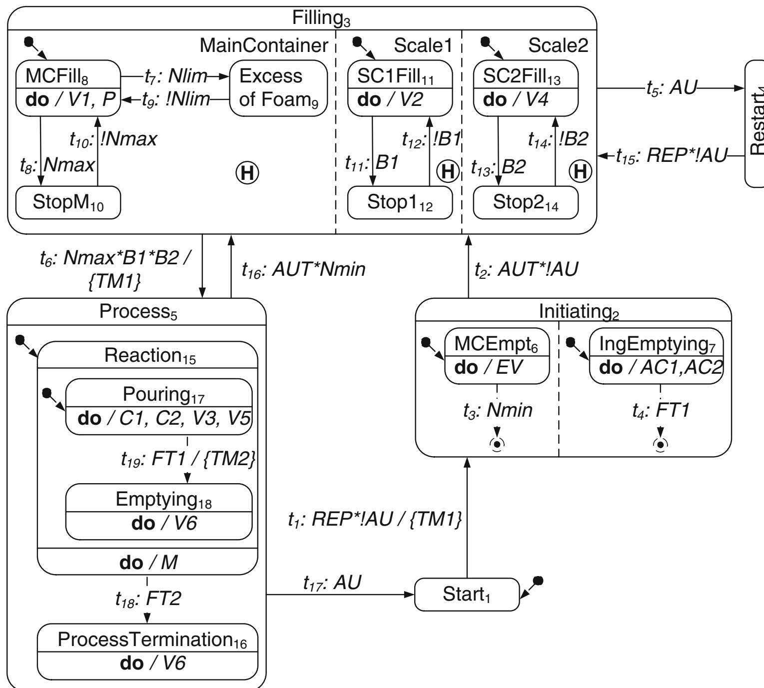


Fig. 10 Statechart diagram for chemical reactor controller

6 Conclusion

UML is a language which can be applied at conceptual modeling stage of digital logic controller design. Designing of logic controller proceeds according to the phases of discrete system development life cycle. Three stages of this cycle are heavily supported by the UML notations. The analysis stage can be supported by class, object and use case diagrams. The design stage is an extended application of standard UML diagrams made by specialized semantic interpretation. Author's semantic interpretation are formulated with the view of synthesis and implementation in the FPGA devices. The diagrams used at this stage are activity, sequence, collaboration and state machine diagrams. The last one is a transition between imprecise UML model and formal description behavior of controller for synthesis and implementation in the FPGA structures.

References

1. Bazydło, G., & Adamski, M. (2011). Specification of UML 2.4 hierarchical state machine and its computer based implementation by means of Verilog. *Przeglad Elektrotechniczny*, *87*(11), 145–149.
2. Bazydło, G., Adamski, M., & Stefanowicz, Ł. (2014). Translation UML diagrams into Verilog. In *7th International Conference on Human System Interactions (HSI)* (pp. 267–271). Lisbon, Portugal.
3. Biliński, K. (1996). *Application of Petri Nets in parallel controllers design*. Ph.D thesis, University of Bristol, Electrical and Electronic Engineering Department, Bristol.
4. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified Modeling Language User Guide* (2nd ed.), Addison-Wesley Object Technology Series. Boston: Addison-Wesley Professional.
5. Borowik, G., Łabiak, G., & Bukowiec, A. (2015). Fsm-based logic controller synthesis in programmable devices with embedded memory blocks. In J. Nikodem & R.Klempous (Eds.), *Innovative technologies in management and science*, Topics in intelligent engineering and informatics (Vol. 10, pp. 123–151). Heidelberg: Springer International Publishing Switzerland, Cham (ISBN: 978-3-319-12651-7).
6. Boutekkouk, F., Benmohammed, M., Bilavarn, S., & Auguin, M. (2009). UML2.0 profiles for embedded systems and systems on a chip (SOCs). *Journal of Object Technology*, *8*(1), 135–157.
7. de Micheli, G. (1994). *Synthesis and optimization of digital circuits*, McGraw-Hill series in electronical and computer engineering. New York: McGraw-Hill Inc.
8. Erjavec, T. (2009). Introducing the Xilinx targeted design platform: Fulfilling the programmable imperative. *White Paper: Virtex-6 and Spartan-6 FPGA*, (306), 6.
9. Fuentes-Fernández, L., & Vallecillo-Moreno, A. (2004). An introduction to UML profiles. *UPGRADE, European Journal for the Informatics Professional*, *5*(2), 5–13.
10. Gajski, D. D., Vahid, F., Narayan, S., & Gong, J. (1994). *Specification and design of embedded systems*. Englewood Cliffs: Prentice Hall.
11. Grobelny, M., Grobelna, I., & Adamski, M. (2012). Hardware behavioural modelling, verification and synthesis with UML 2.x activity diagrams. In *Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems—PDeS 2012* (pp. 109–114). Brno, Czechy.
12. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*, 231–274.
13. Kuschnerusy, D., Brunsy, F., Bilgic, A., & Musch, T. (2012). A UML profile for the development of IEC 61508 compliant embedded software. In *Proceedings of the 6th International Congress and Exhibition—Embedded Real Time Software and Systems, ERTS2 2012*. Toulouse, France
14. Łabiak, G. (2003). *The use of hierarchical model of concurrent automaton in digital controller design*. Ph.D thesis, Warsaw University Of Technology, Faculty of Electronics and Information Technology, Warsaw, May (in polish).
15. Łabiak, G. (2015). HiCoS Homepage. http://www.uz.zgora.pl/~glabiak.
16. Łabiak, G., & Borowik, G. (2010). Statechart-based controllers synthesis in FPGA structures with embedded array blocks. *International Journal of Electronics and Telecommunications*, *56*(1), 13–24.
17. Łabiak, G., Adamski, M., Doligalski, M., Tkacz, J., & Bukowiec, A. (2012). UML modelling in rigorous design methodology for discrete controllers. *International Journal of Electronics and Telecommunications*, *58*(1), 27–34.
18. OMG, (2006). 250 First Avenue, Needham, MA 02494. UML profile for system on a chip (SoC), U.S.A. August.
19. Object Management Group: OMG, (2011). 250 First Avenue, Needham, MA 02494. Modeling and analysis of real-time embedded systems, June, U.S.A. UML Profile for MARTE.

20. OMG, 250 First Avenue, Needham, MA 02494, U.S.A., April. This version (2.4.1) has been formally published by ISO as the 2012 edition standard: ISO/IEC 19505-1 and 19505-2.
21. Wood, S. K., Akehurst, D. H., Uzenkov, O., Howells, W. G. J., & McDonald-Maier, K. D. (2008). A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. *IEEE Transactions on Computers*, *57*(10), 1357–1371.
22. Zwoliński, M. (2004). *Digital system design with VHDL* (2nd ed.). Upper Saddle River: Prentice Hall.

Index

A
Action (of interpreted Petri net), 3, 5
Action (of UML activity diagram), 135, 143–150, 155, 158–160
  dynamic, 143, 146–150
  indivisibility of, 135, 143–145
  state oriented, 143, 146–150
Action (PRALU), *see* operation of action
Action (SFC), 6, 8
Active-HDL, 89, 158
Activity diagram, *see* diagram
All programmable system-on-chip (APSoC), 15, 16, 19, 24, 28
ALU (language), 7
Analysis
  behavioral, 173, 175
  functional, 172, 174
  object-oriented, 172, 173
Atomization, 63
Automaton
  finite, *see* Finite State Machine
  parallel, 5, 32, 36, 42
  sequent, 32, 36, 37, 42

B
BDD, *see* decision diagram
Behavioral analysis, *see* analysis
Blanket algebra, 107
Boolean function, 1, 5, 42, 104–106, 113

C
C (language), 16, 23, 25, 27, 59
C#, 99
CAD, 101, 123–126, 168, 173, 175, 177
Chain (PRALU), 7, 8, 32–34, 38, 41, 42
Chemical plant, 10, 173
Class diagram, *see* diagram
Closed loop control, 169
Collaboration diagram, *see* diagram
Coloring
  of graphs, 112, 113, 120
  of Petri net, 67–69, 73, 76, 83, 84, 96
Communicating FSM, *see* finite state machine
Communication diagram, *see* diagram
Computation tree logic (CTL), *see* logic
Conceptual modeling, 165, 171
Concurrency, 1, 31, 132, 137, 171, 175
Concurrent FSM, *see* finite state machine
Consensus, 69, 73, 75
Control flow, 135
Controlled object, 134, 147–149, 169
Controller, *see* logic controller
Control memory, *see* memory
Control system, *see* system
Control unit, v, 45, 46, 50, 54
  with datapath, 169
Counterexample, 69, 162

D
Deadlock, 18, 75
Decision diagram, 4, 37, 42
Decomposition, 45, 46, 78, 89, 96, 101, 103, 104, 106, 110, 115–117, 119, 120, 123, 124
  architectural, 94, 95
  functional, 46, 94, 103, 104, 106, 109, 110, 112

© Springer International Publishing Switzerland 2016
A. Karatkevich et al. (eds.), *Design of Reconfigurable Logic Controllers*,
Studies in Systems, Decision and Control 45,
DOI 10.1007/978-3-319-26725-8

symbolic, 106, 109, 110, 113–115, 127
parallel, 93–97, 99, 100
structural, 46, 93, 100, 104
Deduction, 68, 69
Diagram (UML), v
activity, 60, 132, 135, 136, 139, 141, 143, 144, 146, 149, 150, 153–156, 158–160, 162, 168, 171, 173
behavior, 132
class, 99, 172
collaboration, 173
communication, 138, 142
object, 132, 172
sequence, 137, 138, 173
state machine, 2, 60, 135–137, 141, 143, 153, 166, 170, 171, 173
structure, 132
timing, 132, 137, 140, 141
use case, 132, 134, 135, 141, 172
Digital system, *see* system
Digital system design, 31, 50, 68, 77, 89, 101, 103, 131, 166, 167
Droplet size, 57, 58, 63–65

E
Embedded memory block, *see* memory block
Encoding, 75, 78, 83, 84, 87, 89, 96, 100, 101, 103–111, 113, 118, 124, 127
binary, 104, 107, 108, 110, 114, 124, 127
one-hot, 82, 85, 89, 93, 96, 105, 124
partial, 106

F
Finite state machine (FSM), v, 1, 5, 17–20, 22–24, 28, 39, 46, 47, 87, 103–118, 120, 123–127, 131, 135, 165, 170, 171
communicating (CFSM), 16–20, 22, 28
concurrent, 4, 78, 171
hierarchical concurrent (HCSM, HCFSM), 2, 68, 77, 89, 171
hierarchical (HFSM), 16–20, 22–24, 28
parallel hierarchical (PHFSM), 16–20, 22, 25, 27, 28
parallel (PFSM), 19, 20
Firing set, 8, 34, 38, 41
Flip-flop, 32, 37, 38, 41, 82, 85, 88, 89, 93, 98, 104, 105
Formal method, 32, 144, 153
Formal reasoning, 67, 71, 89
Formal synthesis, *see* synthesis
Formal verification, 154, 155, 158, 162, 172
FPGA, v, 1, 4, 9, 16, 32, 37, 42, 45–54, 57, 59, 77, 78, 89, 93, 94, 98, 100, 101, 103–106, 113, 115, 124, 127, 149, 153, 155, 162, 166
Functional analysis, *see* analysis
Fusion of parallel places, 84
Fusion of series places, 84

G
Gentzen logic, *see* logic
GRAFCET, 1, 3, 6, 7
Graph
incompatibility, 112, 113, 118–123
reachability, 31, 69, 71, 72
transition, 38, 40, 41
Graph coloring, *see* coloring

H
Hardware accelerator, 19, 25
Hardware description language (HDL), 4, 50–52, 77, 88, 93, 98, 123, 124, 167
Hardware implementation, *see* implementation
HiCoS, 173, 177
Hierarchical FSM, *see* finite state machine
Hierarchy, 1, 5, 9, 16, 19, 20, 28, 136, 145, 146, 171, 175
Hyperedge, 71, 72, 75
Hypergraph
concurrency, 67–69, 71, 72
sequentiality, 71–75

I
Implementation, 8, 16, 19, 24, 45–48, 51–54, 59, 99–101, 103, 109, 115, 117, 118, 124, 177
circuit (hardware), 4, 8, 9, 31, 32, 36–38, 42, 51, 52, 54, 93, 98, 104–106, 108, 127, 177
logical, 51, 52
Integer linear programming (ILP), 72
Interrupt, 16–18, 22, 24

J
Java, 16, 59, 60

L
Label (PRALU), 7, 8

Ladder diagram (LD), 1, 3, 4, 170
LeonardoSpectrum, 32, 35–37, 42
Life cycle, 172
Linear temporal logic (LTL), *see* logic
Liveness, 68, 75, 76, 89, 161
LOCON-2, 31, 32, 34–36, 42
Logic
  computation tree (CTL), 161
  Gentzen, 77, 78, 80
  linear temporal (LTL), 161
Logical control algorithm, v, 1–9, 32
  parallel, 1, 4, 5, 31, 32, 38, 42
Logic controller, v, 1–4, 6, 9, 15, 16, 28, 45–47, 50–54, 57, 59–63, 67, 69, 71, 76–78, 83, 88–90, 93, 100, 131, 133, 134, 137, 138, 140–150, 153–156, 159, 160, 162, 165, 169, 170
  design of, 67, 68, 90, 134, 141–144, 149, 153, 154, 162, 170
  microprogrammed, 45–54
  reconfigurable (RLC), 15–18, 28, 45, 54, 57, 59, 65, 67, 77
  with object, 169
Logic equation, 98, 100
Logic minimization, 104, 105
Logic synthesis, 45, 54, 77, 93, 101, 103–106, 109, 110, 127, 157, 177
Look-up table (LUT), 23, 47–49, 89, 94, 104, 115, 117, 123–127

M
M2vs, 157, 158, 162
Macronet, 76, 83, 84
Macroplace, 5, 6, 81, 83–85, 87–89, 94, 96, 97, 99
Marking, 3–6, 68, 71, 72, 80–83, 98
  initial, 4, 5, 8, 68, 97
Memory, 16, 18, 22, 23, 32, 45–47, 52, 62, 85
  control, 45–48, 50–52, 54
Memory block, 23, 24, 28, 45, 47, 52, 54, 95
  embedded, 94, 98, 101
  RAM (BRAM), 47, 49, 50, 52–54
Model checking, 9, 60, 69, 90, 150, 153–155, 158, 161, 162
Modularity, 16, 19, 28
Multi-level computing system, *see* system

N
Non-functional requirements, *see* requirements
NuSMV, 69, 155, 158, 162

O
Object diagram, *see* diagram
Object-oriented analysis, *see* analysis
Operation (PRALU)
  of action, 7, 8, 32–34, 41
  of waiting, 7, 8, 32–34, 41
Optimization, 19, 36, 37, 42, 87, 106, 158

P
Paradigm shift, 166
Paradigms, 165, 171
  concurrency, 165
  hierarchy, 165
Parallel FSM, *see* finite state machine
Parallel hierarchical graph-scheme (PHGS), 15, 20, 22, 26, 28
Parallelism, v, 2, 16, 19, 20, 28
Petri net, v, 1–6, 8–10, 60, 67–85, 88, 89, 93–96, 98–101, 103, 135, 171
  α-, 5, 8
  control, 3, 5, 9, 77–79, 89, 144, 146, 150, 153–156
  hybrid, 9
  interpreted, v, 3–6, 9, 68, 77, 78, 89, 153–156, 170
    asynchronous, 5
    Mealy type, 5
    Moore type, 5, 96
    synchronous, 5
  live, 68, 75
  reactive, 9
  real time colored (RTCP), 9
  safe, 4–6, 8, 68, 81
PLA, 4, 32, 37, 42, 105
Place (of Petri net), 3–6, 8, 68, 69, 71, 72, 74, 75, 78, 80–85, 87, 89, 93–101, 155–158
  marked, 4–6, 68, 71, 72, 75, 95
PLC, *see* logic controller
PNSF, 4, 9, 82
Port
  general-purpose (GPP), 16–18, 22
  high-performance (HPP), 16, 17, 22, 23
PRALU, 1, 2, 4, 7, 8, 31–34, 36–42
Process
  design, 50, 68, 89, 131, 132, 141, 142, 154, 173, 175
  implementation, 51, 52, 106, 173
Processing system, *see* system
Profile, 167

R
Rapid prototyping, 9, 85, 88, 158
Reactive system, 146, 169
Reconfiguration, 17, 18, 20, 23, 28, 48, 49, 51, 59, 65, 78, 89
  dynamic, 16, 18, 19, 22, 28
  partial, v, 45–54
  static, 47, 48
Register, 18, 22, 32, 37, 82, 88, 89, 95, 98–100, 104, 105, 169
Requirements, 20, 34, 60, 65, 132, 134, 153, 161, 162, 171, 173, 176
  non-functional, 172
RTL, 35–37, 68, 77, 177
Rule-based logical model, 150, 153–155, 157–162

S
Safety, 57–60, 65, 161
Separation, 112, 113, 115–118, 120
Sequence diagram, *see* diagram
Sequent, 68–70, 74, 75, 77, 81, 85
Sequent calculus, 68, 73, 90
Sequential function chart (SFC), 1, 3, 4, 6–8, 170
Signal, 3, 5, 7, 9, 17, 18, 20, 22–24, 27, 38, 41, 83, 98, 99, 133, 144–149, 169, 174
  control, 15, 17, 169
  input, 17, 18, 38, 145, 148, 149, 156–160, 162
  output, 3, 18, 38, 59, 60, 71, 96, 134, 144–147, 149, 150, 155–160, 162
Siphon, 67, 68, 73–76
Software/hardware architecture, 15–17
Specification, v, 1–3, 6, 7, 9, 20, 27, 31, 50, 57, 60, 67, 68, 79–83, 131, 132, 137, 140, 141, 144, 146, 149, 150, 153–156, 158, 162, 177
  behavioral, 77, 149, 150
  formal, 60, 132, 150, 155, 161, 173
  hierarchical, 76
  Petri net based, 2, 4, 9, 144
  rule-based, 77, 78, 80, 82, 89, 150, 155, 162
  semi-formal, 143, 153
  UML-based, 60, 132, 134, 137, 140, 143–145, 173
Spray, 57–63, 65
SSF file, 177
State, 4, 6, 17, 18, 21, 22, 24, 31, 36, 38, 41, 42, 46, 60, 67, 68, 78, 79, 82–84, 88, 89, 98, 103–110, 113, 116, 118, 123, 124, 127, 132, 135–137, 146–148, 155, 156, 158, 159, 162, 165, 169
  global, 5, 69, 71
  local, 5, 69, 71, 78, 82, 155
State machine diagram, *see* diagram
State machine subnet (SM-subnet), 67, 71, 75, 79, 83, 93–96, 98, 100, 101
State minimization, 103
State space, 73, 75, 78, 83, 90
State transition table, 107, 110, 111
State variable, 106–110, 112, 113, 116, 127
Statechart diagram, *see* diagram (state machine)
Step (of SFC), 6
Structure diagram, *see* diagram
Symbolic reasoning, 68, 77
Synthesis
  logic, 9, 50, 52
Synthesis , *see* logic synthesis
System
  control, v, 9, 10, 50, 57–59, 61, 62, 65, 78, 93, 132, 144, 146–148, 169
    mixer, 133
  digital, v, 31, 88, 93, 103, 108, 131, 141
  multi-level computing, 24
  processing (PS), 15–17, 20, 22–24, 28

T
Tautology, 69
Timing diagram, *see* diagram
Token (of Petri net), 4–6, 8, 68, 71, 80, 81, 89, 135
Transition (of Petri net), 3–6, 68, 71, 74, 75, 78, 80–85, 87–89, 97, 98, 146–149, 156, 158–160
  enabled, 4–6, 71
  firing, 4–6, 80, 81, 89, 93, 94

U
Unified Modeling Language (UML), v, 57, 131, 132, 140–143, 147, 150, 153, 162, 165–168, 171–178
Use case diagram, *see* diagram

V
Verilog, 31, 50, 69, 82, 167, 168
VHDL, 4, 9, 27, 31, 32, 35–37, 42, 50, 69, 77, 78, 81, 82, 89, 90, 99–101, 155, 158, 167, 168, 177

Virtex, 47, 49

X
Xilinx, 166
iMPACT, 169
ISE, 168
XML, 9