UNIWERSYTET TECHNOLOGICZNO-PRZYRODNICZY
IM. JANA I JEDRZEJA SNIADECKICH W BYDGOSZCZY
ZESZYTY NAUKOWE NR 256
TELEKOMUNIKACJA I ELEKTRONIKA 13 (2010) 79-91

PETRI NETS AND ACTIVITY DIAGRAMS IN LOGIC CONTROLLER SPECIFICATION – TRANSFORMATION AND VERIFICATION

Iwona Grobelna, Michał Grobelny, Marian Adamski

Institute of Computer Eng. And Electronics
University of Zielona Góra
Zielona Góra, Poland
{i.grobelna, m.adamski}@iie.uz.zgora.pl, m.grobelny@weit.uz.zgora.pl

Summary: The paper presents formal verification method of logic controller specification taking into account user-specified properties. Logic controller specification may be expressed as Petri net or UML 2.0 Activity Diagram. Activity Diagrams seem to be more user-friendly and easy-understanding that Petri nets. Specification in form of activity diagram may afterwards be transformed into Petri net, which may then be formally verified and used to automatically generate implementation (code). A new transformation method dedicated for event-driven systems is proposed. Verification process is executed automatically by the NuSMV model checker tool. Model description based on specification and properties list is being built. Model description derived from Petri net is presented in RTL-level and easy to synthesize as reconfigurable logic controller or PLC. Properties are defined using temporal logic. In model checking process, verification tool checks whether requirements are satisfied in attached system model. If this is not the case, appropriate counterexamples are generated.

Keywords: formal verification, logic controller, model checking, Petri nets, UML Activity Diagrams

1. INTRODUCTION

Logic controller specification is the first step in the design and development process. It is therefore especially important, that the specification meets user-defined requirements. The specification may be formalized in different forms [12], e.g. by means of Petri Nets or, what may seem more user-friendly, by means of UML 2.0 Activity Diagrams. However, activity diagrams are not well supported by formal verification mechanisms. Nevertheless, they can be transformed into Petri nets which can be then formally verified for consistency between model description and requirements for its behavior. In the article a new transformation method dedicated for event-driven systems is proposed. Activity diagram action nodes are interpreted as Petri net transitions, unlike classical approaches in previous versions of UML where action nodes were interpreted as Petri net places.

Model checking of prepared specification allows to early detect subtle errors resulting from wrong specification interpretation. It is one of formal verification methods among others like e.g. theorem proving [7] [16]. The paper focuses on a new logical model which derives from Petri net and is presented in RTL-level in such a way that it is easy to synthesize as reconfigurable logic controller or PLC

The article is structured as follows. Section 2 presents some background on formal mechanisms needed to specify logic controller behavior, Petri nets and UML 2.0 Activity Diagrams, and a formal mathematical system used in requirements specification. Section 3 concentrates on transformation aspects from activity diagram into Petri net specification. Section 4 focuses on model checking of formal specification in form of Petri net. The article concludes with short summary and future research directions.

2. FORMAL SPECIFICATION BY MEANS OF PETRI NET AND UML 2.0 ACTIVITY DIAGRAM

This section includes some background on formal specification methods by means of Petri nets and UML 2.0 Activity Diagrams, and on temporal logic.

2.1. PETRI NETS

Petri nets [3] [4] [8] [12] were introduced in 1962 by Carl Adam Petri as a general purpose mathematical model for describing relations between conditions and events. They are currently used in many industrial branches for planning and controlling of production flow, design and programming of microprocessor controllers, system software synthesis, etc. There are some design tools available which allow to automatically generate code from Petri net specification [11].

Graphic representation of Petri net can be understood even by non-technical staff. It allows e.g. to specify such behaviors as parallelism and concurrency, choice, synchronization, memorizing, reading or resource sharing [8].

A Petri net is specified by places (represented by a circle) and transitions (represented by a bar or a box) connected together (represented by directed arcs which indicate relation flow, where places can be connected only to transitions, and transitions can be connected only to places). The number of places and transitions is finite and not zero. States are defined by tokens (represented by small full circles, also called markers) inside some places. A transition can be fired only if each of its input places contains at least one token. Then from each of its input places one token is being removed and added to each of its output places.

Control Interpreted Petri Nets [2] specify and model the behavior of concurrent logic controllers and take into account properties of controlled objects. Local states may change after firing of transitions if some events occur. Transition guards are associated with input signals of controller and places are associated with its output signals. Global state of logic controller is built of simultaneously holding local states.

2.2. UML 2.0 ACTIVITY DIAGRAMS

The Unified Modelling Language 2.0 notation [19] simplifies information flow between team members and enables easy understanding of system behavior by non-experienced staff. It was initially introduced for specification, visualization and documentation of software. However, behavioral embedded system design [4] can also be facilitated by using some types of UML diagrams, like activity diagrams, state machines or sequence diagrams.

Activity diagrams are currently used in business domain, modeling of information flow [10] [23] and behavioral software and embedded systems design in software/hardware co-design [20].

Most commonly used parts of UML activity diagrams are action nodes (represented with rounded rectangles with the name of the action inside). The flow of activities is described using lines with arrowheads. Additionally every diagram should start with the initial node (big filled dot) and end with the final point (filled dot with a border). Usually embedded controllers or other discreet systems have parallelism in their description. It is possible to represent it using fork and join nodes (notated by horizontal or vertical bars).

2.3. TEMPORAL LOGIC

Temporal logic [5] [15] [17] introduced into computing science in 1977 by Amir Pnuelli derives from modal logic with possibility and necessity operators. Firstly it was used in concurrent and reactive systems. Currently it is used also in program specification, verification, synthesis and logical programming.

Classical temporal logic is Linear-time Temporal Logic (LTL). It describes relations in the system and state sequences. A formula can change any time its value, e.g. in some states it can be true, and in others it can be false. Basic operators are: *always* (G), *sometimes* (F) and *next* (X).

Temporal logic with time branches is Computation Tree Logic (CTL). Time is presented here as a tree branching out into the future with present moment as the root. Characteristic for branching time logics are path quantifiers: the E operator for some paths and the A operator for all paths. They are for paths beginning from a given state, while state quantifiers are for states in a path. State quantifiers are: the F operator for some states and the G operator for all states. Path and state quantifiers are mostly used together, i.e. $EF\ p$ means that in some paths in some states formula p is true and $AG\ p$ means that in all paths in all states formula p is true.

3. TRANSFORMATION

In this section a new transformation method dedicated for event-driven systems is presented. Firstly, an example is introduced, which will be used to better illustrate proposed transformation method.

As an example to present transformation method from UML 2.0 Activity Diagrams to Petri nets a simple embedded system for movement control of two vehicles [3] has been taken.

Initially, both vehicles are placed at starting points a and c. After pressing the m button, they begin to move to the right simultaneously. If both vehicles reach their ending points (point b for the first vehicle $W1$ and point d for the first vehicle $W2$) vehicle $W1$ returns to its starting point a. Afterwards, the second vehicle $W2$ returns to its starting point c. Then the process can be started again. The real model of described process is presented in Fig. 1.

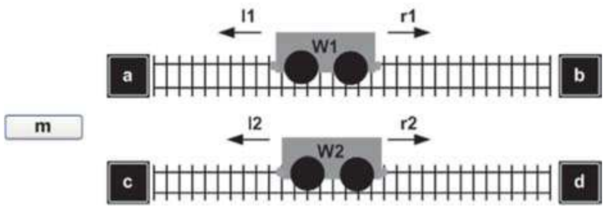

Fig. 1. Real model of analyzed process

The schema of logic controller with input and output signals is presented in Fig. 2. The signals are described in Table 1 and Table 2.

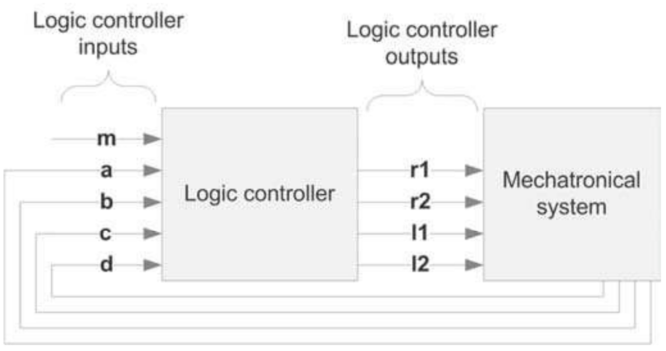

Fig. 2. Logic controller schema

Table 1. Input signals and their meaning

Input signal	Meaning
m	Signal to start the process.
a	The first vehicle *W1* is at its starting point *a*.
b	The first vehicle *W1* is at its ending point *b*.
c	The second vehicle *W2* is at its starting point *c*.
d	The second vehicle *W2* is at its ending point *d*.

Table 2. Output signals and their meaning

Output signal	Meaning
r1	The first vehicle *W1* moves to the right.
r2	The second vehicle *W2* moves to the right.
l1	The first vehicle *W1* moves to the left.
l2	The second vehicle *W2* moves to the left.

Specification of described process, presented in Fig. 3, was prepared by using UML 2.0 Activity Diagrams. Movements to the right are realized simultaneously, while movements to the left (returns) are realized sequentially. Actions are executed when conditions from square bracket are fulfilled (input signals from Table 1 take appropriate values). Values of output signals are specified inside action boxes. They are used for controlling of vehicles movements in the real world (Table 2).

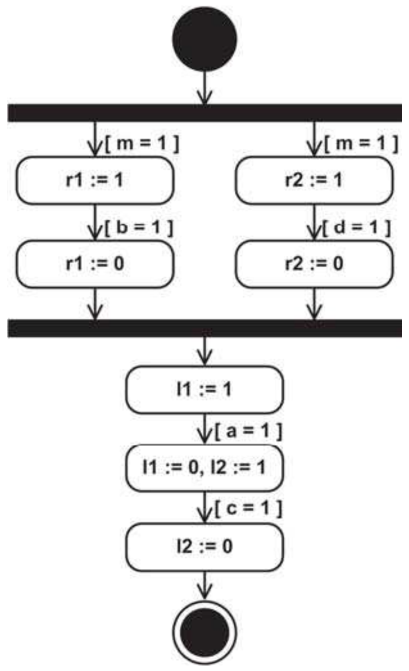

Fig. 3. Activity Diagram for analyzed process

For transformation purposes actions of activity diagrams are treated like transitions [21] [22]. In classical approaches in previous versions of UML action nodes were interpreted as Petri net places. Here, action nodes are interpreted as Petri net transitions. Fork and join nodes (notated by horizontal or vertical bars) are interpreted as fork or join transitions in Petri net. Behavior of controlling process is presented step by step, basing on the interpretation in form of activity diagram. The starting and ending points of activity diagrams refer to appropriate places (*P1* and *P17*) in Petri net. Additionally, input conditions for the actions were treated like decision blocks before actions. Therefore, each input condition in activity diagram is assigned a transition in Petri net with appropriate firing condition. Additional synchronization places (*P10* and *P11*) are necessary, because UML syntax enforces synchronization in join node.

Petri net after direct transformation includes 17 places and 16 transitions (Fig. 4a). However, some places and transitions are redundant. The model compression resulting in reduction of unnecessary delays in circuit performance is executed. The proposed reduction method assumes replacing of transition with condition and transition with action with one transition. This procedure is connected with deleting of unnecessary places. As an example, place *P4* and transition *T4* may be deleted and their etiquettes may be moved to transition *T2*. Transition *T4* reflects assigning of value *1* to signal *r1*. According to the activity diagram (Fig. 3) the action may be executed only if signal *m* is active. It can be therefore assigned to transition which checks the current value of the *m* signal. After reduction of redundant places and transitions Petri net (Fig. 4b) has 10 places and 9 transitions (amount of places and transitions has been decreased by ca. 40%).

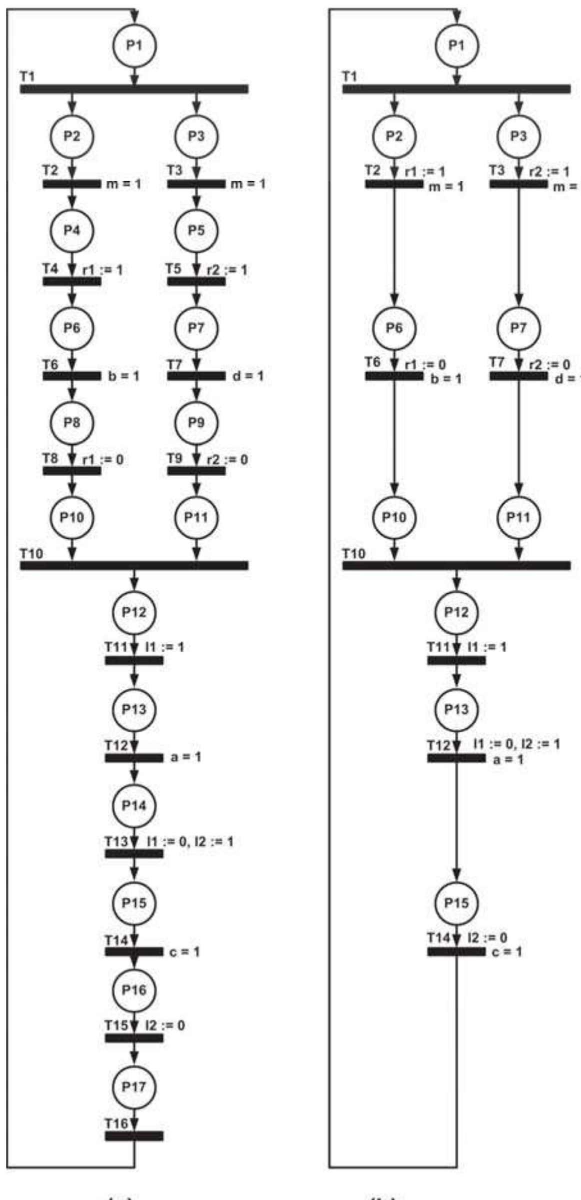

Fig. 4. Petri net after direct transformation from activity diagram (a) and after reduction of redundant places (b)

4. MODEL CHECKING

Model checking technique enables formal verification of logic controller specification. Specification can be checked against behavioral requirements which have to be fulfilled.

First of all, description and requirements list have to be delivered to a model checker tool. A system to be verified should be modeled using the description language of the particular model checker, in our case it is a symbolic model checker NuSMV in the current version 2.4.3 [6] [9]. Requirements list with defined properties should be coded using the specification language of particular model checker. The list should include as many desired properties as possible as only they will be checked. Finally model checker verifies the system and gives an answer whether described model satisfies the specification. In case of detected errors user receives feedback with appropriate counterexamples. What is important is the fact that model checking can be used to verify the whole system or only some part of it. Partial verification is especially valuable in large systems, where the design process is complex and long, as it can be performed step-by-step during the design phase considering each time only a limited subset of requirements.

Design requirements specified by Petri net have to be transformed into the format of the NuSMV model checker. Then the specification can be verified against requirements defined with linear-time temporal logic. Also other specification forms can be formally verified by the model checker tool, an example of algorithmic state machine verification can be found in [13].

The proposed model description derives from Petri net. It is presented in RTL-level (*Register Transfer Level*) in such a way that it is easy to synthesize as reconfigurable logic controller or PLC without any additional changes.

Model description can be prepared as shown in subsections 4.1, 4.2 and 4.3. Requirements list can be defined as presented in subsection 4.4. Subsection 4.5 concentrates on model checking process itself, while subsection 4.6 focuses on the results of performed formal verification.

4.1. VARIABLES DEFINITION

Variables definition (Appendix, lines 2-9) includes global states, input and output signals. Another example of Petri net verification idea is presented in [14] where the verification process is treated more example-specific and focuses rather on controlled objects and its statuses than on global system states.

It is assumed that in one moment only one input signal can be active. Therefore, there is only one variable defined which takes any of possible input signals as its value (the amount of them equals the amount of rows in Table 1 with extra value for no active input signals added).

The number of output signals equals the number of rows in Table 2, where each signal is an independent variable and can be either active or inactive.

Global states are formed of local states from Petri net presented in Fig. 4b. The firing time of transitions is extremely short and only one transition may be fired at particular time unit, what corresponds to one of the important functioning rules of formal model from [1]. However, the nondeterministic automaton with global states of received Control Interpreted Petri Net can be simplified in such a way, that after global state *P2P3*, state *P6P7* will be reached (transitions *T2* and *T3* have the same firing condition so the firing time will be just one after the other and global states *P2P7* or *P3P6* would last the minimum amount of time).

System state is defined by the combination of variable values. In the NuSMV system model of described example, there exists 864 possible combinations, but only 24 of them are available.

4.2. INITIAL VALUES OF VARIABLES

Initially, all variables have some default values assigned (Appendix, lines 11-16). The initial values are changing according to the rules defined next.

4.3. ASSIGNMENT OF VALUES TO VARIABLES

All assignments of next values (Appendix, lines 17-60) happen simultaneously. To make the model simpler, in the sample model description no emergency situations have been taken into account. For example, if a vehicle gets stuck during the movement, it will be necessary to push the vehicles by hand to the starting points and start the process again from the beginning.

To simulate the behavior of real system, values of input signals are chosen randomly, but only expected input signals may be active at particular time.

4.4. REQUIREMENTS LIST

Requirements list may be defined by using either Linear-time Temporal Logic LTL or Computation Tree Logic CTL. The second one is better for verification of nondeterministic programs [18]. We have chosen the first one to define behavioral requirements of the designed logic controller. Among the properties there are some safety properties (situations which must not happen) and liveness properties (situations which have to happen).

Required properties (Appendix, lines 61-82) examine behavior of the designed logic controller mainly after occurrence of some input signals. Let us explain some defined properties. The property in lines 67-68 states that always after the occurrence of *m* input signal, in the next state output signals *r1* and *r2* will be assigned value *1*, what means that both vehicles will move to the right. Next property (lines 69-70) indicates that it should never be the case that both output signals *r1* and *l1* is assigned at the same time value *1*, what means that the first vehicle can not move to the right and to the left (and as the result stay in one place if the forces for moving to the right and moving to the left are equal) at the same time.

4.5. MODEL CHECKING PROCESS

Model description and requirements list are input data for the NuSMV model checker. The tool compares them and generates an answer whether required properties are satisfied in delivered model description.

4.6. RESULTS

After successfully verification a short report is generated (parts of it are presented in Fig. 5 and Fig. 6). It includes list of checked properties with status whether they are satisfied in the model description or not. If any of the requirements is not satisfied, appropriate counterexample is generated. The counterexample presents a trace which may be used by the designer to follow the incorrect situation.

Sample requirements which are satisfied in the described example are listed below (Fig. 5).

```
-- specification  G (input = m ->  X (r1 = 1 & r2 = 1))  is true
-- specification  G !(l1 = 1 & r1 = 1)  is true
-- specification  G !(l2 = 1 & r2 = 1)  is true
-- specification  G (input = b ->  X r1 = 0)  is true
-- specification  G (input = d ->  X r2 = 0)  is true
```

Fig. 5. Satisfied requirements

From the defined requirements a sample property which can not be satisfied (lines 73-74) indicates that always after occurrence of *b* input signal (the first vehicle reached its ending point *b*) finally the output signal *l1* will be assigned value *1* (the first vehicle will finally return to its starting point *a*). This however can not be guaranteed as it may be the case that the second vehicle never reaches its ending point *d* and so the whole system will remain in global state *P7P10* of the Petri net from Fig. 4b. The situation trace is presented in a generated counterexample (Fig. 6).

```
-- specification G (input = b -> F l1 = 1) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
   state = p1
   input = none
   r1 = 0
   r2 = 0
   l1 = 0
   l2 = 0
-> Input: 3.2 <-
-> State: 3.2 <-
   state = p2p3
-> Input: 3.3 <-
-> State: 3.3 <-
   input = m
-> Input: 3.4 <-
-> State: 3.4 <-
   state = p6p7
   r1 = 1
   r2 = 1
-> Input: 3.5 <-
-> State: 3.5 <-
   Input = b
-> Input: 3.6 <-
-- Loop starts here
-> State: 3.6 <-
   state = p7p10
   input = none
   r1 = 0
-> Input: 3.7 <-
-> State: 3.7 <-
```

Fig. 6. Generated counterexample

5. CONCLUSION

Formally specifying system behavior using UML 2.0 Activity Diagrams is an easy and efficient way to document initial negotiation results between customer and supplier.

The specification is then easy-understandable both for engineers and for non-technical partners. On the other hand, Petri nets seem to be better suited for logic controllers and are currently commonly used in the industry. Furthermore, possibility of implementation (code) generation from Petri net specification and insufficient verification methods of UML Activity Diagrams also have to be taken into account. These aspects suggest the solution of transformation from activity diagram to Petri net specification.

Model checking technique is very valuable for formal verification of designed systems. Although it can not prove that the system model is completely correct, it can prove that it has or has not some user-specified desired properties. An advantage is also the fact that model correctness can be verified before the real system physically exists. Therefore it can potentially prevent errors on an early stage of system development.

However, human interaction is especially needed by counterexamples analysis. Generation of them can be caused either by false model description or by invalid requirements specification and every counterexample has to be carefully analyzed.

Future research directions focus on the improvement of transformation from UML 2.0 Activity Diagrams into Petri nets. The aim is to make the transformation fully automatic so that the outgoing specification form is compact and its interpretation corresponds to the interpretation by means of initial activity diagram. Other research directions concentrate on model checking technique and its application to Petri nets specifications. Different transformation methods from Petri net into description format of the NuSMV model checker are being examined.

APPENDIX

Model description with requirements list for discussed example is attached.

```
MODULE main
VAR
   state: {p1, p2p3, p6p7, p6p11, p7p10,
           p10p11, p12, p13, p15};
   input: {none, m, a, b, c, d};
   r1 : {1, 0};
   r2 : {1, 0};
   l1 : {1, 0};
   l2 : {1, 0};
ASSIGN
   init(state) := p1;
   init(input) := none;
   init(r1) := 0;
   init(r2) := 0;
   init(l1) := 0;
   init(l2) := 0;
   next(state) := case
     state = p1 : p2p3;
     state = p2p3 & input = m : p6p7;
     state = p6p7 & input = b : p7p10;
     state = p6p7 & input = d : p6p11;
     state = p7p10 & input = d : p10p11;
     state = p6p11 & input = b : p10p11;
     state = p10p11 : p12;
     state = p12 : p13;
     state = p13 & input = a : p15;
     state = p15 & input = c : p1;
     1 : state;
   esac;
   next(input) := case
     state = p2p3 : {none, m};
     state = p6p7 : {none, b, d};
     state = p6p11 : {none, b};
     state = p7p10 : {none, d};
     state = p13 : {none, a};
     state = p15 : {none, c};
     1: none;
   esac;
   next(r1) := case
     state = p2p3 & input = m : 1;
     state = p6p7 & input = b : 0;
     state = p6p11 & input = b : 0;
     1 : r1;
   esac;
   next(r2) := case
     state = p2p3 & input = m : 1;
     state = p6p7 & input = d : 0;
     state = p7p10 & input = d : 0;
     1 : r2;
   esac;
   next(l1) := case
     state = p13 & input!= a : 1;
     state = p13 & input = a : 0;
     1 : l1;
   esac;
   next(l2) := case
     state = p13 & input = a : 1;
     state = p15 & input = c : 0;
     1 : l2;
   esac;
LTLSPEC
   F (state = p1);
LTLSPEC
   F (state = p15);
LTLSPEC
   G (input = b -> F(state = p7p10));
LTLSPEC
   G (input = m -> X(r1 = 1 & r2 = 1));
LTLSPEC
   G !(l1 = 1 & r1 = 1);
LTLSPEC
   G !(l2 = 1 & r2 = 1);
LTLSPEC
   G (input = b -> F(l1 = 1));
LTLSPEC
   G (input = b -> X (r1 = 0));
LTLSPEC
   G (input = d -> X (r2 = 0));
LTLSPEC
   G (input = a -> X (l1 = 0));
LTLSPEC
   G (input = c -> X (l2 = 0));
```

BIBLIOGRAPHY

[1] Andreu D., Souquet G., Gil T., 2008. Petri Net based rapid prototyping of digital complex system, Symposium on VLSI, IEEE Computer Society Annual, pp. 405- 410.

[2] Adamski M., 2001. A rigorous design methodology for reprogrammable logic controllers, The International Workshop on Discrete-Event System Design, DESDes'01, Przytok, Poland.

[3] Adamski M., Chodań M., 2000. Modelowanie układów sterowania dyskretnego z wykorzystaniem sieci SFC, Wydawnictwo Politechniki Zielonogórskiej, Zielona Góra (in Polish).

[4] Adamski M., Karatkevich A., Węgrzyn M. (ed.), 2005. Design of embedded control systems, Springer (USA).

[5] Ben-Ari M., 2005. Logika matematyczna w informatyce. Klasyka informatyki, Wydawnictwa Naukowo-Techniczne, Warszawa (in Polish).

[6] Cavada R. et al. NuSMV 2.4 User Manual, downloaded from http://nusmv.fbk.eu/

[7] Clarke E.M., Wind J.M. et al, 1996. Formal methods: State of the Art and Future Directions, ACM Computing Surveys, Vol. 28, No. 4.

[8] David R., Alla H., 1992. Petri Nets & Grafcet. Tools for modelling discrete event systems, Prentice Hall.

[9] Eshuis R., 2006. Symbolic Model Checking of UML Activity Diagrams, ACM Transactions on Software Engineering and Methodology, Vol. 15, No. 1, pp. 1-38.

[10] Eshuis R., Wieringa R., 2001. A Comparison of Petri Net and Activity Diagram Variants, Proc. of 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems, pp. 93-104.

[11] Frey G., Litz L., 1998. Verification and Validation of Control Algorithms by Coupling of Interpreted Petri Nets, Proceedings of the IEEE SMC'98, Vol. 1, pp. 7-12.

[12] Gomes L., Barros J.P., Costa A., 2006. Modeling Formalisms for Embedded System Design, Embedded Systems Handbook, Taylor & Francis Group, LLC.

[13] Grobelna I., 2008. Formalna analiza interpretowanych algorytmicznych maszyn stanów ASM z wykorzystaniem narzędzia model checker, Metody Informatyki Stosowanej nr 3, Tom 16, pp. 107-124 (in Polish).

[14] Grobelna I., 2008. Formal verification of logic controller specification using NuSMV model checker, X Międzynarodowe Warsztaty Doktoranckie OWD'2008, Archiwum konferencji PTETIS, Vol. 25, pp. 459-464.

[15] Huth M., Ryan M., 2004. Logic in Computer Science. Modelling and Reasoning about Systems, Cambridge University Press.

[16] Kern C., Greenstreet M.R., 1999. Formal Verification in Hardware Design: A Survey, ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 4, Issue 2, pp. 123-193.

[17] Klimek R., 1999. Wprowadzenie do logiki temporalnej, AGH Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków (in Polish).

[18] Lamport L., 1980. "Sometime" is sometimes "not never", On the Temporal Logic of Programs, Proceedings of the Seventh ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, pp. 174-185.

[19] http://www.omg.org

[20] Schattkowsky T. UML 2.0 – Overview and Perspectives in SoC Design, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05).

[21] Staines T.S., 2008. Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets, 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pp. 191-200.
[22] Tričković I., 2000. Formalizing activity diagram of UML by Petri nets, Novi Sad J. Math, Vol. 30, No. 3, pp. 161-171.
[23] Yen-Liang C., Sammy C., Chyun-Chyi C., Irene C., 2000. Workflow Process Definition and Their Applications in e-Commerce, IEEE 2000, pp. 193-200.

SIECI PETRIEGO I DIAGRAMY AKTYWNOŚCI W SPECYFIKACJI STEROWNIKÓW LOGICZNYCH – TRANSFORMACJA I WERYFIKACJA

Streszczenie

Praca prezentuje metodę formalnej weryfikacji specyfikacji sterownika logicznego uwzględniającą właściwości podane przez użytkownika. Specyfikacja sterownika logicznego może być przedstawiona m.in. w postaci sieci Petriego lub diagramu aktywności języka UML. Diagramy aktywności wydają się być bardziej przyjazne i zrozumiałe dla użytkownika niż sieci Petriego. Specyfikacja w postaci diagramu aktywności może zostać przekształcona do sieci Petriego, która następnie może być formalnie zweryfikowana i wykorzystana do automatycznej generacji implementacji (kodu). Węzły diagramu aktywności konsekwentnie interpretowane są jako tranzycje sieci Petriego, w odróżnieniu od klasycznego podejścia (w starszych wersjach UML) gdzie odwzorowywało się je jako miejsca sieci Petriego. Proces weryfikacji wykonywany jest automatycznie przez narzędzia weryfikacji modelowej. Tworzony jest opis modelu bazujący na specyfikacji oraz lista wymagań. Nowatorskim podejściem jest przedstawienie sieci Petriego na poziomie RTL w taki sposób, że łatwo jest przeprowadzić syntezę logiczną sieci w postaci współbieżnego rekonfigurowalnego sterownika logicznego lub sterownika PLC bez konieczności przekształcania modelu. Wymagania określone są przy użyciu logiki temporalnej. W procesie weryfikacji modelowej narzędzie weryfikujące NuSMV sprawdza, czy model systemu spełnia stawiane mu wymagania. Jeżeli tak nie jest, generowany jest odpowiedni kontrprzykład.

Słowa kluczowe: formalna weryfikacja, sterownik logiczny, weryfikacja modelowa, sieci Petriego, diagramy aktywności UML