Asian Journal Informational Technology 4 (11): 1012-1019, 2005
© Medwell Online, 2005

Model Transformation for Requirements Verification in Embedded Systems

Rachid Boudour and Mohamed T. Kimour
Department of Computer Science, LRI,
University of Annaba, Bp. 12, Annaba, Algeria

Abstract: In embedded systems, use cases have become increasingly popular as a means of requirements specification and drive all the development activities,particularly in validation ones. However, they are usually written in informal text form describing the interactions between the environment and the system. This prevents using formal methods in requirements verification. Though, they are becoming a practical means to ensure the correctness of system models, formal methods still are not commonplace in embedded systems especially in the requirements validation. In this study presenst an approach to model transformation for requirements verification in embedded systems. It firstly consists of transforming the use case structured-text style into an UML activity diagram, which may be reused in the subsequent development steps and secondly we transform this diagram into Pres, a formal notation capable of capturing relevant features of embedded systems. In addition to the offered formal verification framework, we argue that our approach enables enriching the use case model and producing more precise and complete requirements.

Key words: Embedded systems, use cases, requirements verification, petri nets, activity diagram, MDA

INTRODUCTION

Embedded systems re becoming ncreasingly sophisticated and complex, while at the same time experiencing a shorter time to market with greater demands on reliability and security. As a result, the need for systematic software development methods and efficient tools for embedded systems is now greater than ever[1]. In these systems, there is a growing recognition of requirements engineering as the initial and possibly the most important development activity, where the real demands to be placed on the system have to be identified and captured in a consistent and unambiguous manner[2-5].

The most notable UML-based development processes for embedded and real-time systems[3,4,6,7] are based on use cases to capture requirements. Such methodologies suggest that the software development process should be use case driven. That is, use cases are not only used for documenting the requirements, but they also drive other important activities such as requirements validation, design, and testing.

For the level of complexity typical to embedded systems, traditional validation techniques, like simulation and testing, areneither sufficient nor viable to verify their correctness. First, such techniques may cover just a piece of the system model. Second, long simulation times and bugs found late in prototyping phases have a negative impact on time-to-market. Though formal methods have been widely used in software development, they are not commonplace in embedded systems requirements and design, especially in requirements model verification[1].

In this study offers an alternative to traditional validation techniques of embedded systems model, by formalizing use cases to drive ensuring the correctness of these models. Our transformational process (Fig. 1) will remedy to some of the limitations of traditional methods. Furthermore, it will give a better understanding of the system, help uncover ambiguities, and reveal new insights in the system. It uses symbolic model checking, based on a special petri net representation,.called pres (Petri net based Representation for Embedded Systems)[8], a formal notation capable of capturing relevant information characteristics to embedded systems. First, it transforms the informal requirement model expressed by use cases into an UML 2.0 activity diagram. Indeed, use cases are usually written in plain text. Many research[9,10] support that this is usually thebest choice, keeping in mind that one of the main purposes of use cases is the communication between customer and developer. However, text is ambiguous and may contain some inconsistencies and lead to differentinterpretation. The use case textual descriptions are prone to mistakes and incompleteness. Therefore, they cannot be used for automatically checking the requirements model. In contrast, use case text transformation into an activity diagram, not only provides a rigorous process of requirements verification but also helps developers to uncover ambiguities, impreciseness and even omissions that may be present in the use case text. Second, the derived activity diagram is converted into pres using an

Corresponding Author: Rachid Boudour, Department of Computer Science, LRI, University of Annaba, Bp. 12, Annaba, Algeria

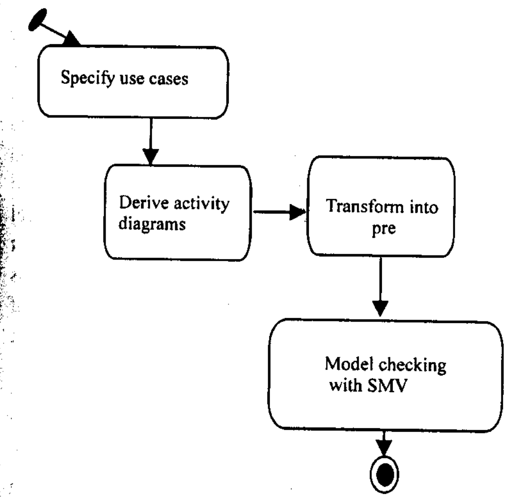

Fig. 1: Activity flow of the approach

appropriate model transformation rules. We have chosen to firstly transform a use case into an activity diagram, because the latter presents a semi-formal nature, suitable to understanding and manipulation by the users and constitutes a precise means to communication between users and developers and then facilitate comprehension and validation of the user requirements. In contrast, Pres is a formal model, difficult to understand and use by the users but also by the developers themselves, however, it is needed as input language to the SMV tool[11] to model check the embedded systems. Our transformation process from activity diagram to Pres is automatically performed, and is therefore transparent to the users. Model checking with SMV Derive activity diagrams Transform into Pres Specify use cases. It is worth noting that activity diagrams established at the requirements step, may be reused at the next steps of development process (analysis, design, implementation and test). They provide both the benefits of an improved requirements specification and the effective means that lead to requirements verification. Once system requirements have been specified by use cases, it is possible, with our approach, to validate embedded system properties as well as timing constraints. Moreover, our approach copes with the model-driven development processes, especially the one proposed by the MDA[12]. In this area, we are witnessing a paradigm shift, where models are no longer mere contemplative documentation, but are used as productive artefacts for analysis, verification and code generation. Model transformation has become central to most software engineering activities. It refers to the process of modifying a (usually graphical) model for the purpose of analysis (by its transformation to some other domain), analysis, refactoring, verification, or even code generation. This framework offers visual and formal techniques based on rules, in such a way that resulting models from the transformation process can be subject to analysis, especially the requirements verification.

MANIPULATED MODELS

It present the manipulated models by our transformation process: use cases, activity diagrams and Pres and highlight important elements needed by the transformation process.

Use cases: A use case is a specific way of using the system by performing some part of the functionality[9]. In embedded systems, an actor can be a human being, a computer system, an external I/O device, or a timer. External I/O devices and timer actors are particularly prevalent in these systems[4]. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behaviour required of the system. As use cases serve as a means of communication between developers and users, they are fundamentally written in simple text. In our transformation process, we are interested in the description of a use case defined by a name, actor, preconditions, postconditions, normal steps, and alternative steps according to Cockburn's template[9].

Use case name: Request elevator.
Context of Use: The elevator system has many elevators that service many users at any one time, taking them from one floor to another.
Primary Actor: User, **Secondary actor**: Floor sensor
Precondition: User is at a floor and wants an elevator.
Postcondition: Elevator has arrived at the floor in response to user request.
Description :

1. User presses an up floor button. System selects an elevator to visit this floor.
2. If the elevator is idle, the system determines in which direction the elevator should move in order to service the new request.
3. The system commands the elevator door to close. After the door has been closed, the system commands the elevator to start moving, either up or down.
4. As the elevator moves between floors, the floor sensor detects that the elevator is approaching a floor and notifies the system.
5. The system checks whether the elevator should stop at this floor. If so, the system commands the elevator to stop.
6. When the elevator has stopped, the system commands the elevator door to open.

7. If there are no other outstanding requests, the elevator stays at the current floor with the door open.

Alternatives:

1a. User presses down floor button to move down. System response is the same for the main sequence.
2a. The elevator is moving, Go to step 4.
5a. Current floor is not in the list of the floor to visit, Go to step 4.
7a. When there are other outstanding requests, Go to step 2.

Quality of Service :

1. Elevator movement must be minimized.
2. Use case response time must be minimized.
3. Doors must be closed before starting any elevator movement

Fig. 2. Textual description of the elevator request use case

To this template we have added a quality of service section in which we describe non-functional requirements (response time, security, cost, accuracy, etc.). Figure 2 shows a typical textual description of a request elevator use case in an elevator control system. A use case can be seen as a tuple <ucName, ucActor, ucPre, ucPost, ucSteps, ucAlt, ucQoS>. ucName is a label that uniquely identifies a use case, ucActor is a primary actor and the secondary actors, ucPre is a set of preconditions, ucPost is a set of postconditions, ucSteps is a set of ordered normal steps, ucAlt a is set of alternative steps, and ucQoS is a set of qualities of services. Each step in ucSteps is a tuple <sNumber, sAction> with sNumber a step number, sAction a set of actions(actor action(s) or system response(s)). An action may also be a branching statement to another step. A normal step may be associated with a set of alternative steps.

An alternative step can be seen as a tuple <altStepNumber, guardCond, altStepAction>, with altStepNumber an alternative step number and guardCond a guard condition on this step, and altStepAction an alternative set of actions. A subset of use case steps in an automated teller machine system may be as follows: {<1, User inserts card>;<2, System Asks for PIN>, <2a, [invalid card]>, <2a1, System emits alarm>, <2a2, System ejects card>}.

UML 2.0 activity diagrams: we introduce the UML 2.0 activity diagram concepts we use to model system level behaviour of use cases. UML 2.0 provides activity diagrams with better constructs (Fig. 3), making them more effective and flexible in describing use cases. They have recently undergone a major revision and redefinition of important concepts like, activities, actions, control and data flows, concurrency, procedure call, and exception handling that are very useful in modeling embedded systems. Moreover, these diagrams are a graphical technique that provides a relatively simple and abstract representation using easy tolearn notation. The obvious advantage of this is that they offer a means of communication between developers and clients, and a valuable tool for requirements elicitation.

An activity describes a logical unit of work. It can be broken down in actions. An action is the smallest unit of work that is not decomposed any further. The sequencing of actions or activities is controlled by control and object flow edges. There are three kinds of nodes: action/activity node, object node, and control node.

An object flow is an edge that can have objects or data passing along it. It models the flow of values to or from object nodes. Activities/actions are joined by edges that represent process flows or events. A decision node can model divergent behaviour based on a Synchronization points may also be defined to illustrate how processing may be carried out in parallel, then synchronized at a point before further activity is undertaken. Input and output parameters can be shown in an activity node. This is done via rectangles that are attached to the activities.

Petri net based model: Pres[8] is an adaptation of Petri nets allowing for capture important characteristics of embedded systems. Some of the features of this model will be illustrated using the example shown in Fig. 4. The net represents an elevator control system as studied by Kimour[13]. Pres[8] is constituted by a finite non-empty set P of places, a finite non-empty set T of transitions, a finite non-empty set I of input arcs, a finite non-empty set T of output arcs, and the initial marking M_0 of the net. Like in classical Petri nets, places are graphically represented by circles, transitions by boxes and arcs by arrows. The elevator control system is modeled in Pres (Fig. 4) where the operations performed in the processes are captured by transitions and the data dependence between them is given by the structure of the net. The transitions have been named after the processes. A marking M is a function that denotes the absence or presence of tokens in places of thenet. The model requires the net to be safe or 1- bounded, i.e. no more than one token is allowed in a place. The marking M_0, for the model of the elevator control system in (Fig. 4) shows P_1 as the only place initially marked.

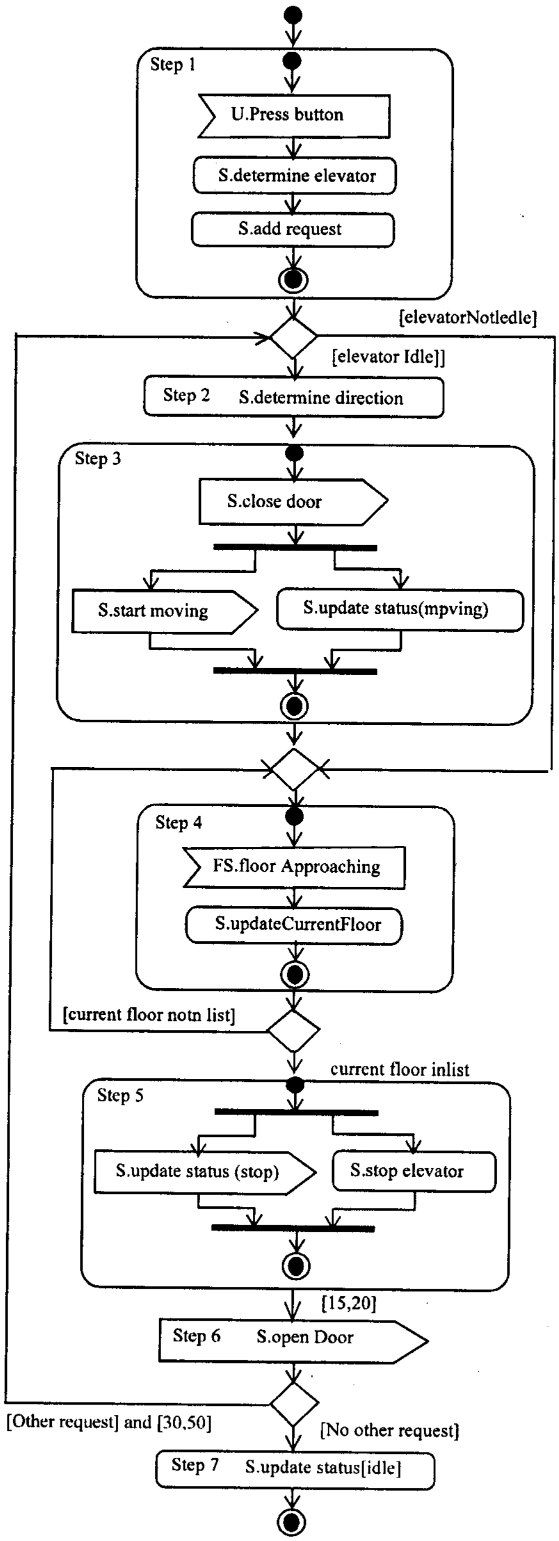

Fig.3: UML activity diagram corresponding to the request elevator use case

In Pres[8] a token is a associated with a pair <v,r> where v is the token value (this value may be of any type) and r is the token time (a non-negative real-valued time stamp). Thus, tokens themselves carry data and time information. There exists atype function t that associates a token type to every place. This is the type of value that a token may vehicle in that place. The token type related to a certain place is an intrinsic property of that place and will not change during the dynamic behaviour of the net. For every transition t, there exists a transition function associated with that transition. Transition functions have as arguments token values of tokens in places of the pre-set of the transition. Pre-set and post-set of a transition t are respectively the set of input places and the set of output places of t.

Transition functions are very important when describing the behaviour of the system to be modeled. They allow systems to be modeled at different levels of granularity with transitions being associated with simple or complex operations. For example, in Fig . 4, there is one transition function associated to transition elevatorSelecting, which defines token values of new token in P_3, when elevatorSelecting is executed.

For every transition t, there exist delay, a non-negative real number, which represents the execution time (delay) of the function associated with that transition. Such a time is captured as transition delay and is inscribed in the respective transition box. In Figure 4, rt represents the execution time of the function associated with the transition elevator selecting. Each transition t in the net may also have a guard G which represents a condition that must be satisfied in order to enable that transition, when all its input places hold tokens. Guards are functions of token values of tokens in the pre-set of a given transition. In Fig. 4, for example, elevIdle represents the condition that must be fulfilled to execute the process elev start moving.

TRANSFORMATION PROCESS FOR MODEL CHECKING

In UML based software development process such as RUP[4], Ropes[6], Comet[4], requirements are expressed by use cases. As use cases are mainly textual description, and to be able to automatically verify requirements, we firstly need to circumvent the drawbacks of the use case text and give rise to important requirement elements such as data and control flows, but also time parameters. To this end, we give here a transformation procedure (Table 1) that allows converting the use case structured text style[9] into an activity diagram.

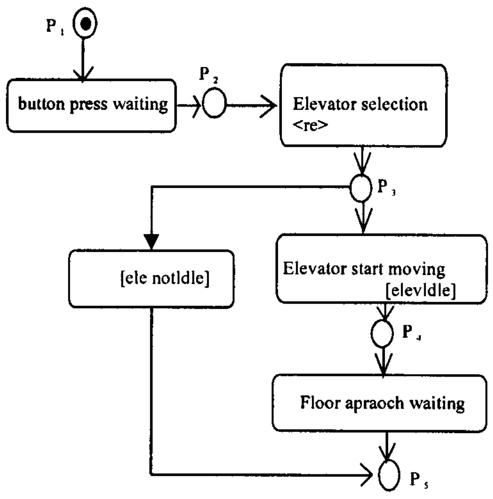

Fig. 4: Pres model fragment of an elevator control system

From use cases to activity diagrams: In the following, we present our procedure to transform a use case text into an activity diagram. Figure 3 shows the resulting activity diagram from the transformation of the use above mentioned use case example. The transformation procedure is composed of two iterations: construction iteration, refinement iteration.

Construction iteration: For each use case, build an activity diagram, where nodes and edges are determined as follows:

- Draw a begin node that corresponds to the use case starting;
- Draw an End node that corresponds to the use case ending;
- Draw a node for each step;
- Draw an edge from the begin node to the node corresponding to the first step;
- Draw an edge between nodes of each two consecutive steps;
- Model the normal flow first, integrate the alternative flows later, and place the steps guards on the corresponding edges.

Refinement iteration: Check the activity diagram to achieve:

- Edges and nodes should be named expressively and consistently, according to the corresponding steps;
- All the necessary nodes and edges should be specified. As the steps are mapped to nodes, missing nodes will emerge andneed to be added.
- All the nodes in the activity diagram should be connected.
- Check the event list created to see if all relevant events are handled and if all the necessary operations are specified in the activity diagram.
- Identify the possible timing constraints and place them on the events that label the corresponding transitions:

From activity diagrams to pres: The model transformation from activity diagrams into Pres is specified by graph transformation rules[14,15]. Graph transformation provides a rule-based manipulation of graph models. A graph transformation rule consists of a Left-Hand side (LHS) graph L, Right-Hand Side (RHS) graph R, and (an optional) application condition N. Informally, L and N of a rule define the precondition while R defines the postcondition. We give in the following the rules to transform the activity diagram representation into pres:

- Every node (except the starting and ending ones) in LHS (the source activity diagram) is modeled with a transition in RHS (the target pres model).
- Every transition in LHS is modeled with a place in RHS.
- Connect every place to the succeeding transition with an arc starting from this place and ending at this transition
- Connect every place (except the starting one) to the preceding transition with an arc starting from this transition and ending at this place .
- Connect the starting place to the succeeding transition with an arc starting from this place and ending at this transition.

PRES BASED MODEL CHECKING

Model checking is an automatic technique for verifying finite-state systems. Specifications are expressed in temporal logic, and the system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications. In the following, we show how the Pres based model is checked against given properties, using the SMV model checker tool.

In Pres, every transition is associated with a behaviour. The behaviour associated with the transition t is defined in terms of its transition function and its transition delay. Unlike the classical Petri net model, each token holds a value v and a time stamp r. When a transition t is fired the marking M will generally change by removing all the tokens from the pre-set and depositing one token into each element of the post-set. A transition t is said to be enabled if all places of its pre-set are marked, its output places, different from the input ones, are empty and its guard is asserted. Tokens, placed into post-set have values and time stamps which depend on the previous tokens in the pre-set and the behaviour of t. When a transition fires, all the tokens in its output places get the same token value and token time. Moreover, every enabled transition has a trigger time r that represents the time instant at the dynamic behaviour of the net.

Based on the above mentioned net, different properties can be studied. For instance, in an elevator control system, the door must not open while the elevator is moving. This sort of safety requirement might be formally proven by checking that the places which represent such a dangerous state are never marked simultaneously. Sometimes, the designer could also be interested in proving that the system eventually reaches a certain state whose marking represents the completion of a task. A given marking, i.e. absence or presence of tokens in places of the net, may represent the state of the system at a certain moment.

This kind of analysis described above, called reachability analysis, is very useful but says nothing about timing aspects nor does it deal with token values. However, in many embedded.systems, time is an essential factor, especially in hard real-time systems (embedded systems are usually real-time), where deadlines should not bemissed, it is crucial to reason quantitatively about temporal properties to assure the correctness of the system model. As a consequence, it is necessary not only to check that a certain state will eventually be reached but also to ensure that this will occur within some time window. As time information is attached to tokens, we can analyze quantitative timing properties: we may, for instance, prove that a given place will eventually be marked in the future and that its time stamp, for any possible condition, will be less than a certain time value that represents a temporal constraint. Such a study will be called time analysis.

A third type of analysis for systems modeled in a Pres model, involves reasoning about values of tokens in marked places. In this model, a place may hold at most one token for a certain marking. Consequently, it is possible to encode markings as boolean functions where the variables correspond to places of the net. Boolean functions can be straightforwardly represented by BDDs[16]. Firing a transition in a Petri net changes the marking into a new one, which is a variation in the state of the system. It is possible to build the BDD that represents the transitionrelation of the system and then compute efficiently the reachable states using BDDs[16]. With such a BDD-based representation we can formally verify properties, specified in CTL[17], using symbolic model checking[17] and accomplish reachability analyses. In our experiments, we use the SMV tool (a BDD-based symbolic model checker)[11] and its input language todescribe and verify systems modeled in Pres.

A program in SMV describes both the system and the specification (properties to verify). The system is described as a collection of modules. Each module may contain variables, its initial state, and assignments of variables for the next state. A process is an instance of a module, in such a way that the model checker executes a step by choosing non-deterministically a process and then executing all assignment statements of that process in parallel.

To translate a Pres model into the SMV input language, we declare each transition as a process that has as parameters its input and output places as well as time stamps of tokens in those places. In the main module we also define the initial marking of the net, assigning initial values to the variables that represent places and to time stamps of tokens in initially marked places. We describeeach transition of the Petri net as a module that adds/removes tokens (changes the marking) when it is executed.

RELATED WORKS

The increasing complexity of embedded systems poses a challenge in verifying their correctness. Recently, some validation approaches for embedded systems have been proposed. Dano *et al.*[18] has proposed a formalization of use cases with Petri nets, he defines a list of temporal relations between use cases (begin at the same time, end at the same time, one after the other, etc.). Alur[17-19], presented model checking procedures based on Hybrid Automata. Balarine[20], presented a verification methodology based on codesign finite state machines that are translated into traditional state automata. This procedure checks if all possible sequences of system inputs and outputs satisfy given properties. In Wimmel[21], approach based on Petri nets, presents a BDD-based model checker for safe nets. Although this approach is intended to verify Petri nets in general, with no particular interest on embedded systems and without dealing with time parameters, it studies different kinds of describing Petri nets. Moreover, it uses the SMV system, developed at Carnegie Mellon University. Another important technique is proposed by Pastor[22]. It is based on Petri nets and uses the efficiency of BDDs to represent sets of markings and reduction rules to transform Petri nets. It can be used for reachability analysis and verification of some properties of Petri nets. Stoy[23], a modeling technique based on Petri nets is proposed, where timed Petri nets with restricted transition rules are used to represent control flow in the system. None of these approaches integrate the formal technique in the use case model. Our approach is to some extent complementary to these existing techniques.

CONCLUSIONS

In this study we have presented a transformational approach for requirements.model verification in embedded systems. The transformation process starts by converting the use case text into an activity diagram and uses the latter as a means to build the input model of an appropriate model checker. The semi-formal nature of UML 2.0 activity diagrams, allows for uncovering ambiguities, omissions, impreciseness, and inconsistency that may be present in the natural language description of the use case. In this way, while preserving the advantages of the use cases' natural language description (expressivity and ease to use), we also allow for using existing tools to verify and prove some properties of embedded systems.

Currently, besides the development of a supporting tool for our approach, we are studied the subject of modifying the XMI DTD to represent our extended activity diagram by means of XML documents, in order to automate the transition between the use case model and the activity diagrams.

REFERENCES

1. Graaf, B., M. Lormans and H. Toetenel, 2003. Embedded software engineering: The state of the practice. IEEE Software, pp: 61-69.
2. Sutcliffe, A.G., N.A.M. Maiden, S. Minocha, D. Manuael, 1998. Supporting scenario-based requirements engineering. IEEE Transaction on Software Engineering, 24: 1072-1088.
3. Jacobson, I., G. Booch and J. Rumbaugh, 1999. The Unified Software Development Process, Addison-Wesley.
4. Gomaa, H., 2000. Designing Concurrent, Distributed and Real-Time Applications with UML. Addison-Wesley.
5. Sommerville I., 2001. Software Engineering, 6th Edn., Addison-Wesley.
6. Douglass, B.P., 2000. Real-Time UML: Developing Efficient Objects for Embedded Systems. Object Technology, Addison-Wesley, 2nd Edn.
7. Maciaszeck, L., 2001. Requirements analysis and system design, Addison-Wesley.
8. Cortés, L.A., P. Eles and Z. Peng, 2000. Verification of embedded systems using a petri net based representation. Proc. Intl. Symposium on System Synthesis.
9. Cockburn, A., 1997. Structuring use cases with Goals, J. Object-Oriented Programming, (part I) and November-December 1997 (part II).
10. Jacobson, I., M. Christerson, P. Jonsson and G. Övergaard, 1992. Object-oriented software engineering: A Use Case Driven Approach, Addison-Wesley.
11. SMV,2005.TheSMVSystemhttp://www.cs.cmu.edu /~modelcheck/smv.html
12. Raistrick, C., P. Francis, J. Wright, C. Carter and I. Wilkie, 2004. Model Driven Architecture with Executable UML. Cambridge University Press. Cambridge,UK. See also the MDA home page at OMG web site: http://www.omg.org/mda/.
13. Kimour, M. T. and D. Meslati, 2004. An approach to building object models with UML in embedded Systems. Intl. J. Com. Inform. Technol. IEE, pp: 12.
14. Varro', D., 'G. Varro and A. Pataricza, 2002. Designing the automatic transformation of visual languages. Science of Computer Programming, 44: pp: 205-227.
15. QVT, 2005. Revised submission for MOF 2.0 Query / Views / Transformations RFP. http://qvtp.org.
16. Bryant, R.E., 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. In: ACM Computing Surveys, 24: 293-318.
17. Alur, R., T.A. Henzinger and P.H. Ho, 1996. automatic symbolic verification of embedded systems, In: IEEE Trans. Software Engineering, 22: 181-201.
18. Dano, B., H. Briand and F. Barbier, 1997. An approach based on the concept of use cases to produce dynamic object-oriented specifications. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97).
19. Hsiung, P.A., 1999. Hardware-software coverification of concurrent embedded real-time systems, In: Proc. Euromicro RTS., pp: 216-223.
20. Balarin, F., H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vincentelli, 1996. Formal verification of embedded systems based on CFSM networks. In: Proc. DAC., pp: 568-571.

21. Wimmel, G., 1997. A BDD-based model checker for the pEP tool, Major Individual Project Report, University of Newcastle, Newcastle.
22. Pastor, E., O. Roig, J. Cortadella and R. M. Badia, 1994. Petri net analysis using boolean manipulation, In: Application and Theory of Petri Nets. Valette, R., (Ed). LNCS 815, Berlin: Springer-Verlag, pp: 416-435.
23. Stoy, E., 1995. A petri net based unified representation for hardware/software co-design. Licentiate Thesis, Linköping University, Linköping.