An Object Oriented Petri Net Language for Embedded System Design

Robert Esser
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland
esser@tik.ee.ethz.ch

Abstract

In this paper a modelling language is described for the automated design of embedded systems. Constraints such as price, performance etc. often imply that embedded systems are implemented as communicating heterogeneous components. Today these components can no longer be classed as software or hardware but as more or less programmable with specialised languages being used to define their functionality and structure. The proposed modelling language consists of a class of high level time Petri nets, augmented with object-oriented mechanisms. It is formal, ensuring unambiguous specification, supports a high level of analysis and is general enough to support other more specialised formalisms. This language constitutes a major part of the CodeSign design methodology [8] developed at the ETH for embedded system design.

Keywords: embedded systems, Petri nets, multiple formalisms, object technology

1. Introduction

Recent gains achieved in efficiency and the mastering of complexity during the design of embedded systems have been due to the introduction of languages and methodologies that add additional levels of abstraction. These enable designers to view the ever increasing system complexity at appropriate levels of detail. However introducing further levels of abstraction results in an increasing gap between system models and their implementation. Embedded systems often have complex behaviour that may not be able to be completely or easily described in a single formalism, so it is desirable to support a number of formalisms that are well suited to modelling particular parts of an embedded system. The modelling language presented in this paper supports the use of multiple formalisms, representing different models of computation within a single system model. This not only allows parts of a system to be modelled naturally but is used to bridge the gap between high abstraction and implementation by providing efficient code generation and system analysis.

Using an object oriented Petri net based kernel language, it is shown how other, more application specific formalisms, can be defined in terms of the kernel language. The advantage of this approach is that the interaction semantics between formalisms are given by the underlying formal kernel language which results in a well defined formal system model. Furthermore embedded systems are often implemented on heterogeneous components, for example general purpose microcontrollers, application specific hardware or specialised programmable hardware (digital signal processors etc.) Cost effectiveness requires that an implementation architecture be chosen that meets a particular set of functional and non functional constraints. The language described here offers mechanisms for design exploration based on flexible system configuration functions and version management.

2. Modelling Language

The modelling language is a combination of object oriented concepts and Petri nets. The Petri net class used is a composition of high level Petri nets [10], [13], [18] and *time* Petri nets [16], [17]. It defines the minimum set of expressiveness needed to describe the functional and behavioural aspects of the target systems. Time plays an important role in all embedded real time systems and any language that purports to model such systems in any level of detail must include a concept of time. Time models for Petri nets and their analysis are described in [3] and [23]. Two basic Petri net models for handling time have been proposed - *time* Petri nets [16], [17] and *timed* Petri nets [22]. *Timed* Petri nets associate a finite firing duration with each transition of a net. The firing rule of these Petri nets is modified to include the time it takes to fire a transition under the assumption that a transition occurs as soon as it is enabled. These Petri nets and the subsequently developed *stochastic* Petri nets [1] have been mainly used for performance evaluation.

Time Petri nets associate a finite enabling duration with each transition in a net. For a transition to occur it must have been continuously enabled for at least the minimum duration and for not more than the maximum duration. Time Petri nets are more general than timed Petri nets, i.e. a timed Petri net can be modelled with a time Petri net, but the converse is not true. Time Petri nets are adequate for expressing temporal constraints. Analysis techniques for formal verification of time dependent behaviour in time Petri nets are described in [3]. More interesting is the technique described in [5] where a compositional approach to the analysis of time Petri nets components is proposed.

Figure 1 shows an example of high level time Petri nets. This class of Petri nets is an extension of standard place-transition Petri nets where tokens carry data and transitions have associated functions and time constraints. Figure 1 shows a model of a system containing four transitions (squares) representing a process fork, concurrency and synchronisation. The transitions are inscribed with tuples (α, β) representing time constraints, guard functions $g_x(in)$ that define additional enabling conditions and functions $f_x(in_1, in_2)$ to determine the type and values of tokens produced as a result of a transition occurrence.

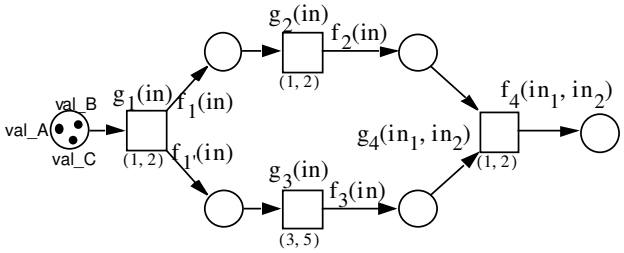


Figure 1. A high level time Petri net where tokens represent data and transitions contain time intervals and functions.

The use of high level time Petri nets as a kernel language has several advantages:

- intuitive graphical representation;
- simple formal definition, reducing the possibility of ambiguous interpretation of system models;
- operational semantics that support simulation and execution;
- analysis techniques that prove a number of interesting structural properties;
- ability to be analysed for temporal properties;
- successful use in many different application areas.

2.1. Combining Object Oriented Concepts and Petri Nets

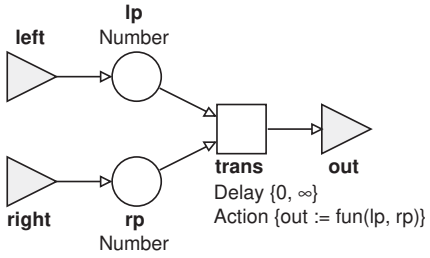
The absence of both powerful structuring primitives and composition mechanisms in Petri nets is still a major weakness [12]. Various forms of graphical folding mechanisms [7], [20] have been proposed but only aid in information hiding and not in the composition of nets. Modularisation, including composition, proposals include those described in [2], [9] (which also presents module refinement techniques) and Hierarchical Coloured Petri Nets [13].

Recently there have been attempts to combine the structuring techniques of object oriented programming with Petri nets [6], [14], [24]. Allowing behaviour to be described with a Petri net enables complex concurrent behaviour to be modelled together with the data that it manipulates. In high level Petri nets data is modelled as tokens residing on places. Thus classes can define functions, executed within transitions, and a Petri net where (token) data and behaviour are encapsulated. However as discussed above this is not enough, as it is also necessary to support code (net) reuse and sharing. This, in an object oriented context, implies some form of inheritance and polymorphism.

Most existing object oriented Petri net approaches only consider token data to be object oriented. The approach chosen in this paper is similar to that taken in LOOPN++ [14] where not only the token data is object oriented but also the Petri net itself. Conceptually every element is an object and can be used in every context. Hence token data are objects that may also contain a Petri net, enabling multi level systems to be modelled e.g. an operating system controlling a number of processes. Two major differences to LOOPN++ are the addition of a concept of time and the concept of system instantiation, where a system is instantiated to represent a particular configuration in a step preceding the start of execution.

In the object oriented approach taken, every object is an instance of a particular class. An extension to the high level time Petri nets defined above is the addition of a type system, i.e. the values of tokens on places are constrained to be of a particular type. In this approach a type is synonymous with a class hierarchy, therefore class refinement also implies type refinement. This is an important mechanism in refining a model so that it can ultimately be efficiently implemented on a heterogeneous target architecture. For example, a filter can be described as a data flow graph transforming numerical data. The performance requirements may require that the filter be implemented on a DSP. As many DSPs only efficiently handle numbers of fixed bit length a further refinement could involve defining the number of bits used to represent

this data and how an overflow is handled (truncation, rounding or exception).




```
FUNCTION fun (a,b:Number):Number
"a function template"
{<ABSTRACT>
   RESULT := 0
}
```

Figure 2. Showing the abstract superclass of classes *Adder* and *Multiplier*. It defines the behaviour and state with the above Petri net and also an abstract function *fun*. Interfaces are represented as triangular shaped elements.

2.2. Components

In the object oriented approach systems are modelled as communicating components or objects. Components have behaviour which is modelled with the kernel language or another formalism described with the kernel language. Components communicate by sending and receiving objects to and from one another. Components are hierarchically structured, i.e. contain other components, which leaves scope during the instantiation process to decide which particular component will be inserted. This mechanism allows design exploration and configuration to be supported in a convenient way. Unlike other approaches where only special communication objects may be sent, communicating standard objects allows for introduced interfaces to wrap protocols around objects if necessary.

Figure 3 shows a component that implements the function $f(a, b) = res = 2a + b^2$ and contains three embedded components using the object oriented high level time Petri net kernel language. Components have defined interfaces over which objects are communicated. Here it is possible to imagine a partitioning that places the three embedded components on different implementation target devices and different configurations with implementation specific components that represent this partitioning. The model of $f(a, b)$ does not assume sequential semantics, as depending on how the system is partitioned, a schedule is determined that ensures that it is functionally correct and also meets its performance constraints, for example it can be seen that a schedule that allows pipelining could be used.

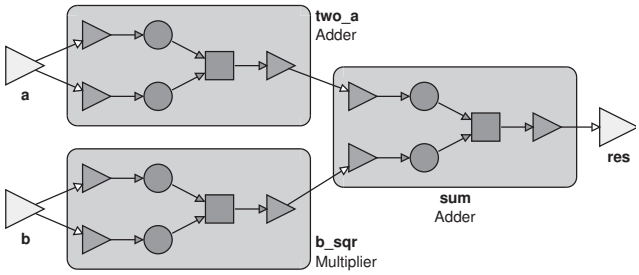


Figure 3. Graphical representation of a component containing embedded components.

3. Support for Other Formalisms

An embedded system design methodology should allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled. The approach proposed in this paper is based on a formal object oriented kernel language and concepts from graph grammars [19], [21]. Graph grammar productions are defined which specify the editing rules (syntax) of a formalism. The semantics are modelled using kernel language components to represent the elements of a formalism. Multiple formalisms communicate with one another with semantics defined by the kernel language. An advantage of this approach is that tools need only consider the kernel language rather than each formalism individually, therefore only one simulator, code generator and refinement mechanism etc. is needed. It is however still possible to capitalise on formalism specific tools and methods, for example, for efficient code generation.

As all components are instances of a class it follows that any formalism assembled from components is also fully object oriented. Thus all the benefits of code reuse, abstraction and refinement provided by the object oriented kernel language are also applicable to all derivative formalisms. Figure 4 shows the model architecture used to implement the ideas presented in this paper.

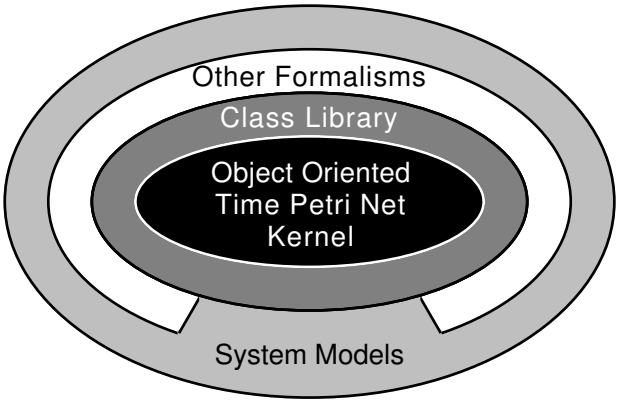


Figure 4. The model architecture.

4. Describing a Formalism

The following steps illustrate how a new formalism is supported. These steps use the expressiveness of the kernel language in which elements of a formalism and editing rules are described:

1. The syntax rules of a new formalism are represented as graph grammar productions. Productions define how elements of a formalism are edited to create models, e.g. the insertion of an element or the connection of two elements. These productions are called Syntax Graph Grammar (SynGG) productions [21] and are described completely in terms of the new formalism;

2. An object oriented time Petri net component class is associated with every element in a formalism. These component classes may be already existing classes where an element of a formalism is directly represented by a kernel language component or they may be new classes representing other behaviour. The resulting class hierarchy represents the elements of a formalism and should reflect that elements may have similar behaviour, which can be inherited. Care must also be taken in how communication between the proposed formalism and its environment is defined - often a class will represent a part of a model described wholly by a single formalism and in which only classes representing the elements of a formalism may be instantiated. These classes can be defined abstractly and, through subclassing, have the ability to be refined and reused. Thus all formalisms are object oriented and models may use the refinement techniques afforded by the object oriented approach;

3. Complementing every SynGG is a Semantic Graph Grammar (SemGG) production which describes the semantics of the syntax defined by the SynGG. These semantics are defined in terms of the kernel language (i.e. object oriented time Petri nets) and involve the component classes defined to represent the formalism's elements.

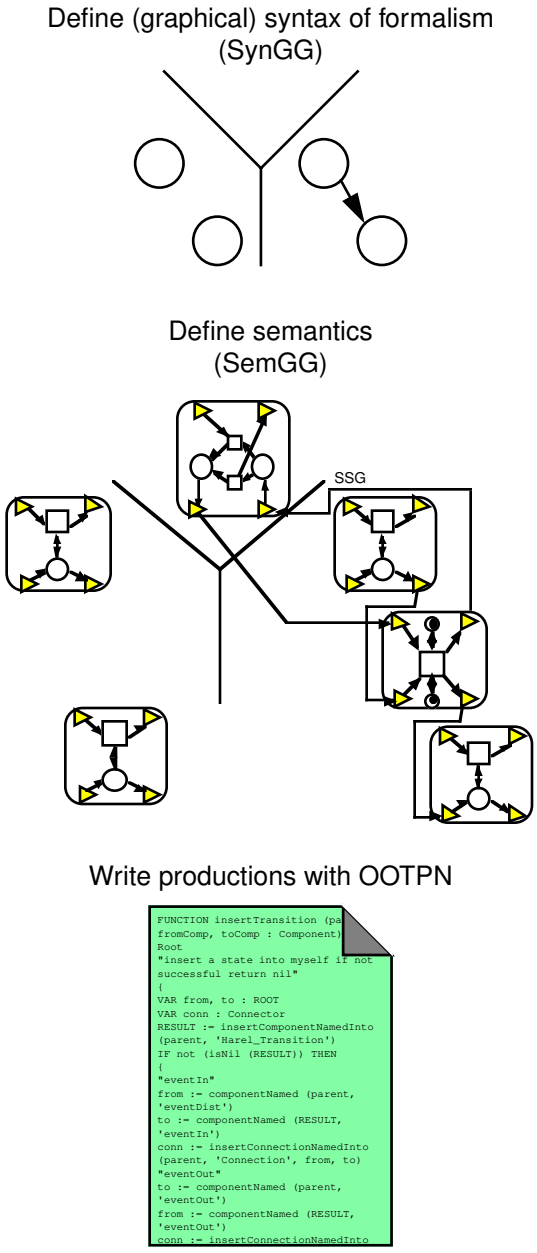


Figure 5. Syntax, Semantic and implementation Graph Grammar productions describing the addition of a state-transition in a simple Finite State Machine formalism.

4. The productions defined by the SemGG are more complex than those represented by the SynGG as they describe how kernel components are composed to represent the semantics of a formalism including the syntax (editing rules) represented by the SynGG.

5. Functions are then written representing the productions defined by the SemGG and are associated with a particular class in the object oriented environment, i.e. they are

defined for the class in which a production can be executed. These functions represent the conditional arcs and the actual subgraph selection and replacement mechanisms;
6. To complete the description of a formalism, functions are created that define how an element of a formalism is graphically represented (shape, weight etc.) including how its current state is displayed. In addition functions are created that define the interface to the editor. This allows a formalism specific editor to be generated in which actions cause previously defined productions to be executed.

A formalism is contained within a component representing the (sub) system model. This component has kernel interfaces through which the formalism communicates with other parts of the model described by the kernel language or other formalisms. Therefore the class that defines the component also defines the editing facilities (productions) that are appropriate for the formalism. In the case of synchronous data flow (SDF) graphs [15] it defines productions to insert SDF nodes and interfaces (used to communicate with other components) and to connect SDF nodes with one another.

Figure 6 shows a component with an equivalent functionality as that in figure 3. It can be seen that the SDF representation is better suited to describe the required functionality. The use of an appropriate formalism can significantly simplify the representation of a component. The round elements are SDF nodes representing an operation that can be carried out when sufficient input data is present. An embedded component in figure 3 shows a Petri net representation of a SDF node.

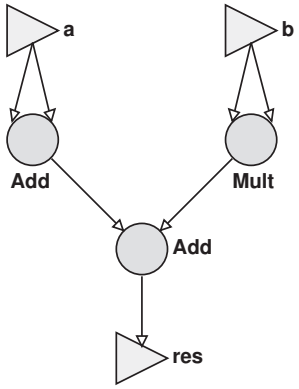


Figure 6. Synchronous data flow (SDF) representation of a component containing embedded components with an equivalent functionality as that in figure 3.

7. Other Formalisms in an Object Oriented Environment

As described previously, other formalisms are defined using the kernel language. This kernel language is object oriented, facilitating the creation of classes which are instantiated in a number of contexts. Any element of a formalism described as a component of a class can be subclassed. As with the components of the kernel language, a subclass may be extended and refined in a number of ways. This is an important feature of the object oriented approach and when applied correctly results in considerable gains in productivity [4]. In object oriented systems results are reused by either subclassing or embedding.

The subclassing of elements of a formalism allows a refined element to be used in place of the standard one. Kernel language inheritance rules ensure that the refined element will always be able to be used correctly. Subclassing may be prevented by defining a class to be monomorphic. This ensures that only suitable classes are able to be refined. Assuming that the functions representing SemGG productions allow a choice of component classes to be instantiated, subclassing is available to all formalisms. This enables abstraction and refinement to be used.

Embedding is a formalism specific feature and usually represents hierarchical structuring. Formalisms that do not support hierarchy, such as SDF, cannot make use of embedding at the formalism level. In contrast StateCharts [11] does support hierarchy and can benefit from the advantages of code sharing.

The following figures show an example of subclassing and reuse using the StateCharts formalism and implemented using the object oriented time Petri nets kernel language. Three classes are used to build a state machine able to count to 4 and 8. Figure 7 shows a subclass of *Abstract_Sequential*. This class implements a simple two bit counter. It does not define a state-transition from state *b1* and state *b0* as this would prevent any subclass from implementing a larger counter.

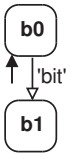


Figure 7. The class *Bit_2* implementing a 2 bit counter.

Figure 8 extends the 2 bit counter to 4 bits. It is a *subclass* of *Bit_2* and has been enhanced with the addition

of two states (*b2*, *b3*) and 3 state-transitions. The states and transitions inherited from the class *Bit_2* are shown as being shaded. The state-transition from state *b3* to state *b0* produces an event named *'nibble'*. This event is broadcast throughout the state machine and, in addition, is made available through the external event interface from where it can be used by other parts of a model, possibly modelled with different formalisms. Class *Bit_4* reuses the states and state-transitions defined in class *Bit_2*.

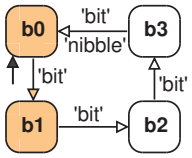


Figure 8. The class *Bit_4* implementing a 4 bit counter.

Finally in figure 9 the class *Bit_8* is shown to consist of two *embedded* components of class *Bit_4*, one of which is marked as the initial state. As the two component hierarchical states are instances of the same class they only differ in their dynamic state information. Ignoring the internal states of both components of class *Bit_4* the system can be regarded as only having 2 states (*b_0_3*, *b_4_7*) and 2 state-transitions that occur when event *'nibble'* is present. The state-transition from state *b_4_7* to state *b_0_3* produces the event *'byte'* to indicate that 8 bits have been counted.

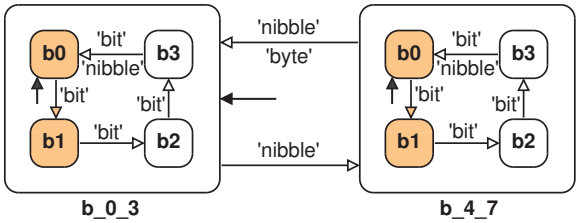


Figure 9. The class *Bit_8* implementing a 8 bit counter.

Figure 10 shows part of the class hierarchy used to implement the 3 classes above.

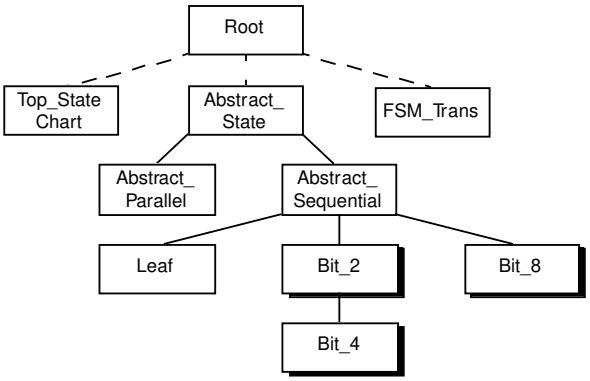


Figure 10. Class hierarchy tree for StateCharts classes implementing an 8 bit counter.

Figure 11 shows part of the instantiation tree for a component of the class Bit_8. This component can be embedded within another class or used directly within a subclass of *Top_StateChart* where a production will connect the event mechanism to the component. This figure does not show the kernel components used to implement the elements of the StateCharts formalism. The components marked *inherited* are defined in the parent class of the defining class.

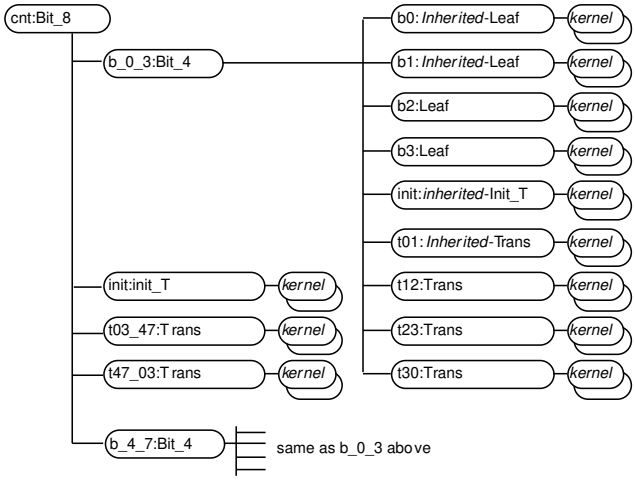


Figure 11. Instantiation tree of the class *Bit_8* (figure 9).

8. Conclusions

This paper presents a new approach to the design of embedded systems. This approach is based on a formal kernel language augmented with object oriented concepts. The formal kernel language is a class of high level time Petri nets which offers the following benefits:

- intuitive graphical representation;
- simple model of concurrency and synchronisation;
- operational semantics that support simulation and execution;
- time model that directly supports the modelling of real time systems;
- proven analysis techniques.

The inclusion of object oriented concepts provides facilities for creating components that encapsulate behaviour and state, and thereby supports:

- abstraction;
- refinement;
- encapsulation;
- reuse and sharing.

Components are described with classes that define how inheritance properties are propagated and provide mechanisms for configuration, reuse and sharing. A system is a composition of communicating components at every level of abstraction. During the design of an embedded system classes are defined, refined, reused and configured. Refinement is supported by substituting less refined components with components containing an increased amount of detail. Abstraction and refinement allow complex systems to be modelled and successively refined towards an implementation.

The use of this class of Petri nets allows relevant aspects of embedded system design to be addressed. These include:

- the use of application specific formalisms;
- the ability to parameterise or configure models;
- the support for design exploration.

Other approaches use a number of domain specific formalisms with which different aspects of a systems behaviour are modelled. These formalisms may be wholly formal methods or a combination of formal methods and informal ones. They often neglect the question of the semantics of the combination of formalisms within one specification.

The proposed approach uses only a single object oriented time Petri net kernel language for modelling both systems and formalisms. In addition the semantics of combining multiple formalisms are completely given by this underlying kernel language. The flexibility supported by the formal language is available to all other kernel based formalisms as they are also object oriented and can be refined and reused in the same manner as the kernel language.

9. References

[1] Ajmone Marsen M., Balbo G., Conte G., Donatelli S., Franceschinis G., Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing, 1995

[2] Bernardinello L., De Cindio F., A Survey of Basic Net Models and Modular Net Classes, Lecture Notes in Computer Science 609, Springer Verlag 1992

[3] Berthomieu B., Diaz M., Modeling and Verification of Time Dependent Systems Using Time Petri Nets, IEEE Transactions on Software Engineering Volume 17 Number 3, 1991

[4] Booch G., Object Oriented Analysis and Design With Applications, The Benjamin/Cummings Publishing Company Inc. Redwood City, Calif. 1994

[5] Bucci G., Vicario E., Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets, IEEE Transactions on Software Engineering Vol. 21, No. 12, 1995

[6] Christensen S., Toksvig J., DesignBeta V2.0.1 BETA Code-Segments in CP-nets, University of Aarhus, Internal report, 1993

[7] Dähler J., Ein Werkzeug für den Entwurf verteilter Systeme auf der Basis erweiterter Petri-Netze, DISS. ETH Nr. 8770. PhD Thesis Swiss Federal Institute of Technology, 1989

[8] Esser R., An Object Oriented Petri Net Approach to Embedded System Design. PhD Thesis ETH Zurich 1996, VDF Verlag Zurich, 1997.

[9] Fehling R., A Concept of Hierarchical Petri Nets with Building Blocks, In Lecture Notes in Computer Science 674, Advances in Petri Nets 1993, Springer-Verlag, 1993

[10] Genrich H.J., Lautenbach K., System Modelling with High-Level Petri Nets, Theoretical Computer Science 13, North Holland, 1981

[11] Harel D., StateCharts: A Visual Formalism for Complex Systems, Science of Computer Programming 8 (1987) 231-274 North Holland, 1987

[12] Jensen K., Coloured Petri Nets: A High Level Language for System Design and Analysis, Advances in Petri Nets 1990, G. Rozenberg (ed), LNCS 483, Springer-Verlag, 1990

[13] Jensen K., Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use - Volume 1: Basic Concepts, EATCS Monographs in Computer Science, Vol. 26, Springer-Verlag, 1990

[14] Lakos C.A., Keen C.D., LOOPN++: A New Language for Object-Oriented Petri Nets, Department of Computer Science, University of Tasmania, 1994

[15] Lee E.A., Messerschmitt D.G., Synchronous Data Flow, Proceedings of the IEEE, Vol 75 No. 9, September 1987

[16] Merlin P., A Study of the Recoverability of Computer Systems, Thesis, Department of Computer Science, University of California, Irvine, 1974

[17] Merlin P., Faber D.J., Recoverability of Communication Protocols, IEEE Transactions on Communications, Vol. Com-24, No. 9, September 1976

[18] Murata T., Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, Vol 77, No. 4, 1989

[19] Nagl M., A Tutorial and Bibliographical Survey on Graph Grammars, Bericht Nr. 69/78 Abteilung Informatik, Universität Dortmund, 1978

[20] Oswald H., Esser R., Mattmann R., An Environment for Specifying and Executing Hierarchical Petri Nets, 12th International Conference on Software Engineering, Nice, France, 1990

[21] Pezzé M., Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1994

[22] Ramchandani C., Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Massachusetts Institute of Technology, Project MAC, Technical report 120, February 1974

[23] Starke P. H, Analyse von Petri-Netz-Modellen, B.G. Teubner Stuttgart, 1990

[24] Verkoulen P.A.C., Integrated Information Systems Design: An Approach Based on Object Oriented Concepts and Petri Nets, University of Technology: Eindhoven PhD Thesis, 1993