Applying an Object-Oriented Petri Net Language to Heterogeneous Systems Design

Robert Esser

ESEC SA
Cham, Switzerland
robert.esser@esec.com

Jörn W. Janneck, Martin Naedele

Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
{janneck, naedele}@tik.ee.ethz.ch

In the first part of this paper a modeling language is presented for the design and evaluation of heterogeneous systems which are comprised of components with different characteristics, e.g. control-dominated parts and data or material flow dominated parts. The language consists of a class of high level time Petri nets, augmented with object-oriented mechanisms. It is formal, ensuring unambiguous specification, it supports a high level of analysis and it is general enough to support other more specialized formalisms. The second part presents some of our experiences in the application of this language to a real world modeling task in an industrial setting.

1 Introduction

A *heterogeneous system* is a complex entity made up from subsystems with different characteristics. Typically, such a system will be composed of parts which are mainly control dominated and parts which model the flow of data or material, as well as parts which combine these aspects. Often these systems are highly concurrent, and the interactions between subsystems can be highly complex and very different in nature.

Recent gains achieved in efficiency and the mastering of complexity during the design and modeling of heterogeneous 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.

Heterogenous systems often show complex behavior that frequently cannot be completely or easily described in a single formalism, so it is desirable to support a number of formalisms that are well suited to modeling particular parts of an embedded system. The modeling language presented in this paper supports the use of multiple formalisms, representing different models of computation within a single unifying formalism, thus saving the need to specifically describe interactions between components modeled in different formalisms and allowing an unified analysis of the entire system - while retaining the option of looking at specific components in their 'native' description method.

This Petri net based modeling language is presented in section 2. The mechanisms for embedding other formalisms within it are sketched in section 3, with special emphasis on their interactions with the overall object-oriented structure of our approach.

In section 4 we will present some insights and results from an industrial application of the proposed method and tool and point out typical challenges in this area and how they were approached using our methodology.

2 Modeling Language

The modeling language is a combination of object-oriented concepts and a class of temporal Petri nets. The Petri net class used is a composition of high level Petri nets [11], [14], [19] and *time* Petri nets [17], [18]. It defines the minimum set of expressiveness needed to describe the functional and behavioral aspects of heterogeneous systems. Time plays an important role in all such systems and any language that purports to model them in any level of detail must include a concept of time. Time models for Petri nets and their analysis are described in [3] and [25]. Two basic Petri net models for handling time have been proposed - *time* Petri nets [17], [18] and *timed* Petri nets [23]. *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, e.g. 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 is an example of high level time Petri nets which are used as the kernel language. 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 system containing four transitions (squares) representing a process fork, concurrency and synchronisation. The transitions are inscribed with tuples (a, b) representing time constraints, guard functions $g_i(...)$ that define additional enabling conditions and functions $f_i(...)$ to determine the type and values of tokens produced as a result of a transition occurrence.

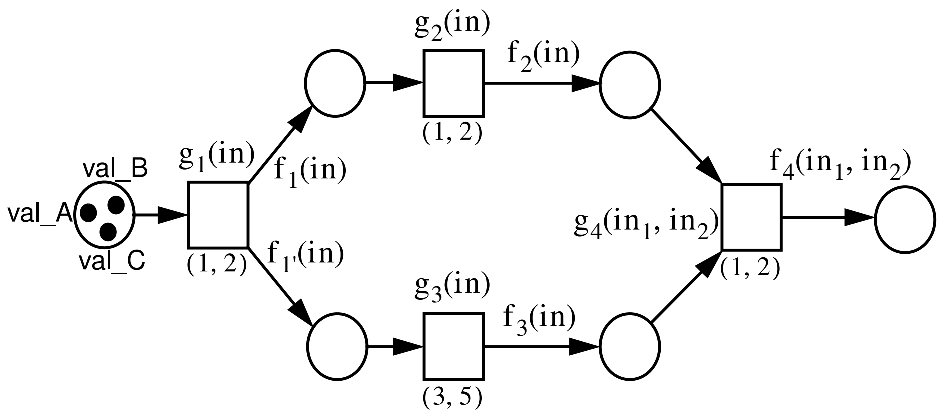


***Figure 1.* A high level time Petri net where tokens represent data and transitions contain timing intervals and functions**

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.

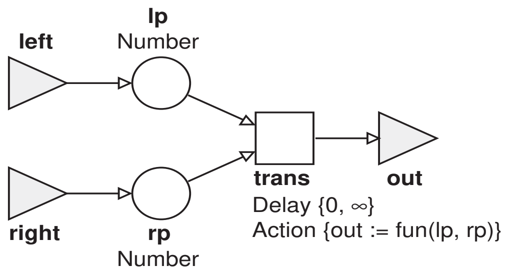
2.1 Combining Object-Oriented Concepts and Petri Nets

The absence of both powerful structuring primitives and composition mechanisms in Petri nets is still a major weakness [13]. Various forms of graphical folding mechanisms [7], [21] have been proposed but they only aid in information hiding and not in the composition of nets. Modularization including composition proposals include those described in [2], [10] (which also presents module refinement techniques) and Hierarchical Colored Petri Nets [14].

Recently there have been attempts to combine the structuring techniques of object-oriented programming with Petri nets [6], [15], [26]. Allowing behavior to be described with a Petri net enables complex concurrent behavior to be modeled together with the data that it manipulates. In a high level Petri net data is modeled as tokens residing on places. Thus classes can define functions, executed within transitions, and a Petri net where (token) data and behavior are encapsulated. However as discussed above this is not sufficient, 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++ [15] 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 modeled, 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. For the types of systems targeted by the approach described in this paper, time is a critical component and must be expressable.

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, type is synonymous with class, therefore class refinement also implies type refinement. This is an important mechanism in refining a model, which facilitates top-down modeling of large and complex systems. For example, one component of a technical system might be a signal filter, which can be described as a data flow graph transforming integer data to integer data. A refinement may involve defining the precision used to represent this data and how overflows or other exceptions are handled.




```
FUNCTION fun (a,b:Number):Number
"a function template"
{
 error('should not be executed')
 RESULT := 0
}
```

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

2.2 Components

In the object-oriented approach systems are modeled as communicating components or objects. Components have behavior which is modeled with the kernel language or another formalism described in terms of the kernel language. Components communicate by sending and receiving objects to and from one another. Components are hierarchically structured, i.e. they may 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 convienient way. Unlike other approaches where only special communication objects may be sent, communicating standard objects allows 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. They can be refined by subclassing them and adding new elements to describe the refined behavior. While interfaces may be added in derived classes, inherited interfaces may not be removed, ensuring that derived classes possess at least the interfaces of all their superclasses and thus allowing for their polymorphic use.

3 Supporting other Formalisms in an Object-Oriented Environment

As mentioned above, other formalisms are defined in terms of an object-oriented Petri net kernel language, facilitating the creation of class templates 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 from either subclassing or embedding.

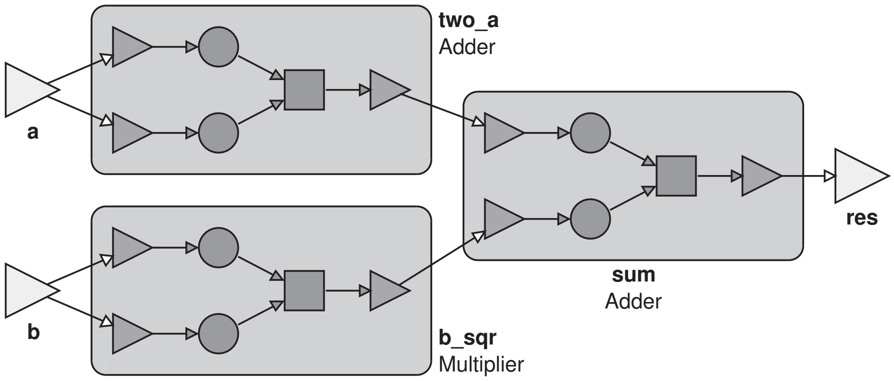


Figure 3. **Graphical representation of a component containing embedded components.**

Subclassing elements defined in a certain formalism allows refined elements to be used in place of standard ones, thus creating a form of polymorphism. The kernel language inheritance rules ensure that the refined element will always be able to be used in the correct way by requiring a derived class to have at least the interfaces of its superclass. Subclassing may be prevented by defining a class to be monomorphic. This ensures that only suitable classes are able to be refined.

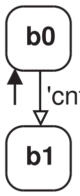


Figure 4. **The Class *Counter_2***

Embedding is a formalism-specific feature implementing a hierarchical structuring based on *containment*. Formalisms that do not support this concept, such as Synchronous Data Flow (SDF) graphs, cannot make use of embedding for the behavioral description of components. In contrast, StateCharts [12] do 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 4 shows a subclass of *Abstract_Sequential*. This class implements a simple 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 5 extends the counter to count to 4. It is a *subclass* of *Counter_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 *Counter_2* are shown as being shaded. The state-transition from state *b3* to state *b0* produces an event named *'cnt4'*. 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 using different formalisms. Class *Counter_4* reuses the states and state-transitions defined in class *Counter_2*.

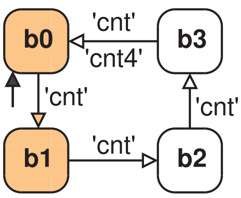


Figure 5. **The Class *Counter_4***

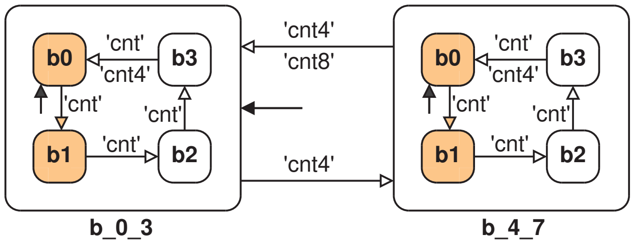


Figure 6. **The Class *Counter_8***

Finally, in figure 6 the class *Counter_8* is shown to consist of two *embedded* components of class *Counter_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 *Counter_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 *'cnt4'* is present. The state-transition from state *b_4_7* to state *b_0_3* produces the event *'cnt8'* to indicate that 8 bits have been counted.

Figure 7 shows part of the class hierarchy used to implement the three classes above. Figure 8 shows part of the instantiation tree for a component of the class *Counter_8*. This component can be embedded within another class or used directly within a subclass of *Top_State* where the event mechanism will be connected 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.

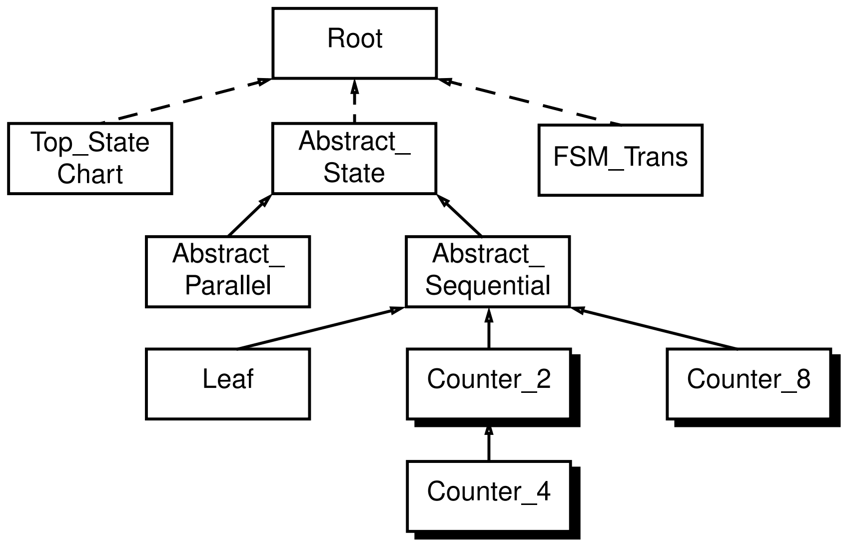


Figure 7. **Class hierarchy tree for StateCharts classes implementing a counter that counts to 8**

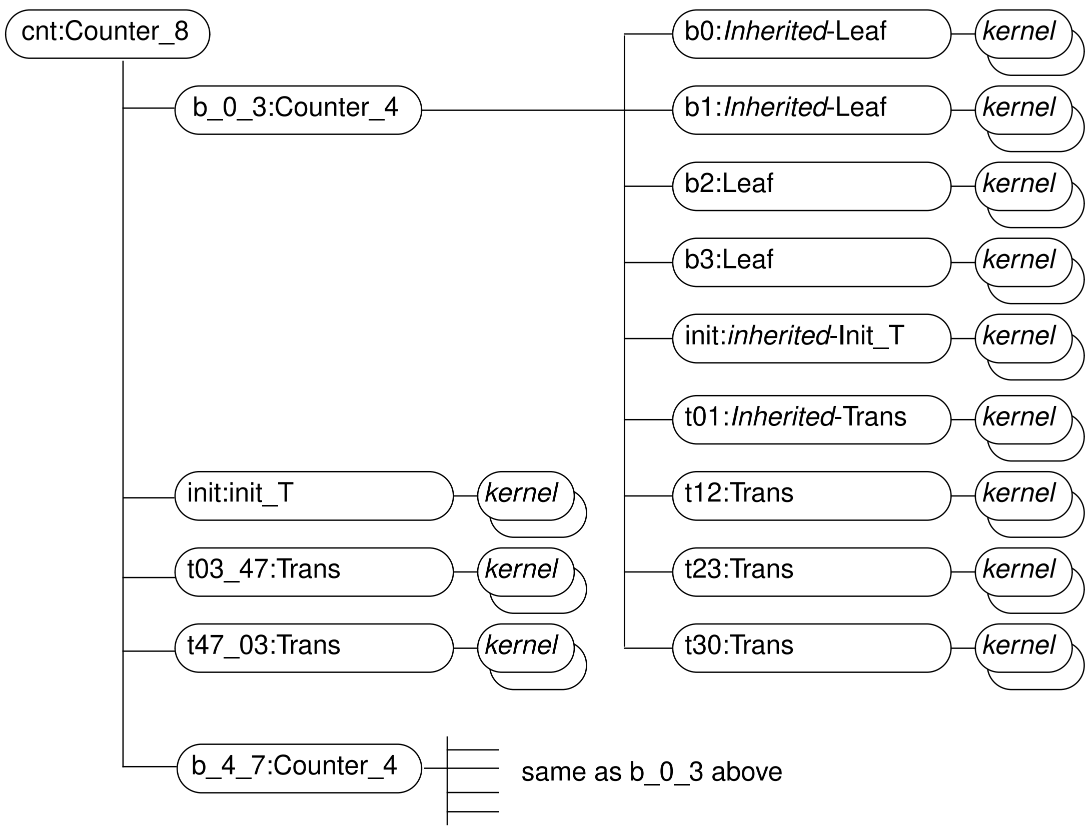


Figure 8. **Instantiation tree of the class *Counter_8***

4 Case Study: Die Bonder

4.1 Motivation

As part of the a cooperation between ESEC SA, a leading manufacturer of semiconductor assembly equipment, and the Computer Engineering and Networks Laboratory of the ETH we applied the object-oriented time Petri net (OOTPN) language and the CodeSign tool to creating and evaluating a model of a die bonding machine.

This case study had two objectives. First, to obtain a detailed timing model of the ESEC die bonder, that could be used for making units-per-hour (uph) calculations/predictions, based on individual customer's die and leadframe material and particular die bonder configurations (which are numerous and diverse in nature). It was also intended to be utilized in an ongoing effort to improve the product design towards higher uph values. Second, we were interested in applying our tool and approach on a real world example and environment, thus obtaining application experience and further requirements for improving the methodology and the functionality of the CodeSign tool for real world engineering tasks.

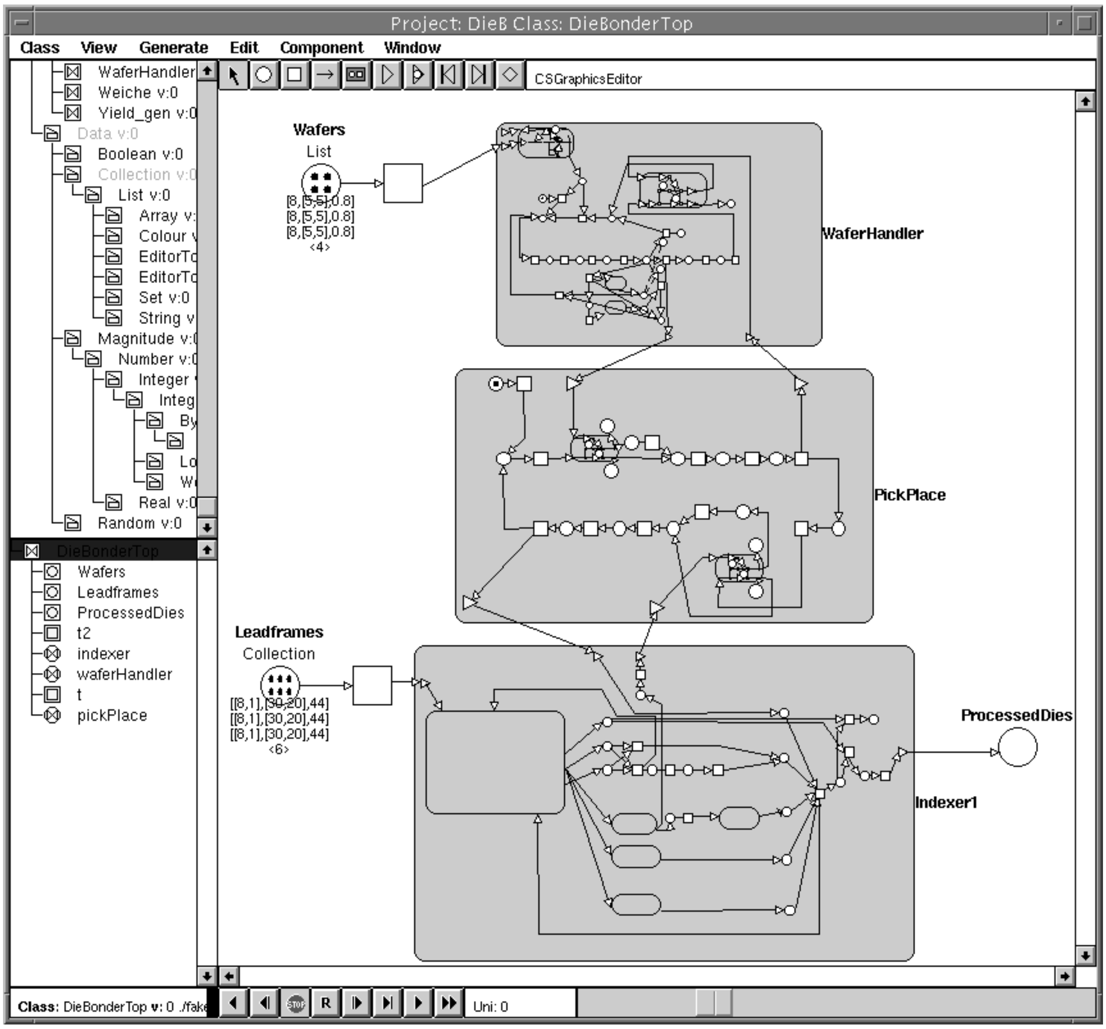


Figure 9. **Die bonder top view**

4.2 Description of the Model

The ESEC die bonder is a highly complex and heterogeneous system, containing hardware, software and mechanical components with a very high degree of concurrency. The model incorporates both control flow and data/material flow through the system.

Figure 9 shows the top view of the model with its three main components: The wafer handler loads the wafer and positions the dies underneath the pick-up point. The indexer loads and transports the leadframes between the glue dispensing and bonding positions. The pick-and-place arm moves the dies between wafer table and bonding position. The input tokens represent wafers (wafer diameter, die length, die width, yield) and leadframes (number of pads in x and y, pad width and length, and number of leads per pad).

Figure 10 shows part of the indexer component in more detail. For each camera several components representing alternative image processing algorithms are available. Each alternative is derived from a camera top class that defines the interface to the surrounding parts of the system. It is therefore possible to configure the die bonder model with different image processing options without having to change the model's structure: Each composite component can contain either other composite components, leaf components or functions that hand back a component as their return value. In the case of the indexer shown here, the cameras are embedded as and instantiated by functions (see Figure 11) which allows the required flexibility with respect to different system configurations.

The cameras are also examples for the embedding of different formalisms into the OOTPN language. In figure 12 it can be seen that the component "postbondOBC_SDF", which is mainly an image processing algorithm, can be adequately modeled with the synchronous data flow formalism and can be seamlessly integrated into the OOTPN environment.

The CodeSign tool supports the proposed OOTPN methodology of inheritance and composition. Two list windows on the left side of the editor window (figure 9) give access to all the component classes currently loaded into the system (top) and allow navigation through the composition hierarchy of the model (bottom).

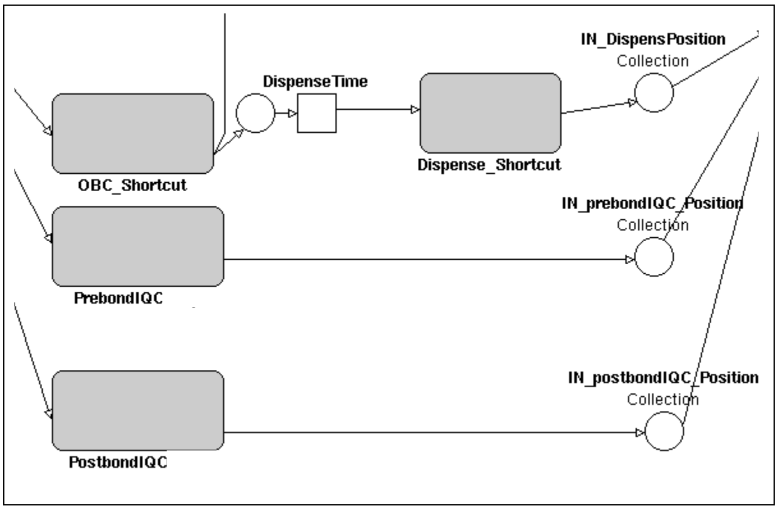


Figure 10. **Camera components embedded in the indexer**

4.3 Results

The graphic nature of Petri net formalism supports an intuitive understanding of the processes described with it and therefore enabled the software, hardware, and process specialists involved to work together and incorporate their specific domain knowledge into a common model. The hierarchical component structure further supports this, as it permits a top-down refinement process during which the domain experts could successively replace the initial behavioral description with more detailed representations of the actual system.

The object-oriented aspects of the OOTPN language used considerably increases the productivity of the modeler, as components could be reused within the model and for models of related products. The component class structure allows polymorphism (which enforces superclass interface compatibility in subclasses) in conjunction with being able to represent embedded components by generating functions for easy configuration of system options on model instantiation, thereby avoiding the overhead of having to keep several almost identical models up-to-date. The version management functions of CodeSign also facilitate easy model development.

Prior to the case study described here a spreadsheet based model already existed that calculated only the production time for one cycle. A cycle in this context means getting one die from the wafer, transporting it to the corresponding pad of the leadframe, bonding the die onto the leadframe, and advancing both indexer and wafer to the next position. Multicycle events like wafer and leadframe magazine changes were only included as corrections on the single cycle result.

In addition the spreadsheet model could not handle conditional dependencies between consecutive cycles. Also the spreadsheet representation is not very well suited to describing the complex concurrency and synchronization issues found in this particular problem domain. Spreadsheets are in general hard to debug which resulted in a couple of mistakes that were only found when the spreadsheet model was compared in detail with the CodeSign model.

The modeling of multicycle events requires a detailed representation of the wafer, that is, of the random distribution of defective dies and of the geometric distribution of dies on the wafer. The generation of this detailed representation from the physical sizes of wafer and die (input values) can comfortably be handled by the programming language that the OOTPN language uses to denote transition functionality.

```
FUNCTION postbondIQCactivated : Boolean
{
     RESULT := true
}

FUNCTION postbondIQCTwoActivated : Boolean
{
     RESULT := false
}

FUNCTION configPostbondIQC : PostbondCamera
"Instatiate camera according to algorithm options"
{
##'/optimise'##
IF postbondIQCactivated () THEN
     {
     IF postbondIQCTwoActivated () THEN
           {
                 RESULT := PostbondIQCTwo.new ()
           }
     ELSE
           {
                 RESULT := PostbondIQC_SDF.new ()
           }
     }
ELSE
     {
           RESULT := Postbond_Shortcut.new ()
     }
}
```

Figure 11. **Configuration and instantiation functions of the indexer component**

The CodeSign simulation environment also allows simulations to be animated. Hence the tool can be used by design engineers to explore interdependencies between system components in addition to marketing and field service staff being able to quickly obtain characteristic numerical predictions of throughput numbers based on particular customer configurations. Direct comparison between production cycle times obtained from the simulation and actual values measured on the machine show that the simulation is only about 0.5 percent off the real value measured on the machine, compared with a 3 percent deviation in the spreadsheet based model.

From the above case study it can be seen that using the OOTPN language and the CodeSign tool permits a much more accurate and detailed model of a heterogeneous system to be developed compared to the previous spreadsheet approach in about the same time. The developed model also has not even come close to the limits of both the tool and the OOTPN language. Improvements in the facilities for gathering, analyzing and consolidating simulation data will be the next points to be addressed by our work.

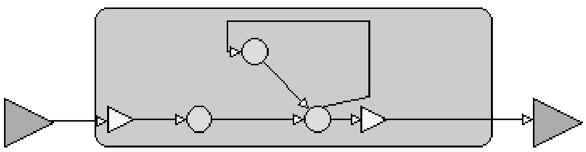


Figure 12. **Component uses SDF formalism**

5 Conclusions

In the first part of this paper an approach to the modeling and design of heterogeneous systems was proposed 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 synchronization;
- operational semantics that support simulation and execution;

- time model directly supporting the modeling of real time systems;
- proven analysis techniques.

The inclusion of object-oriented concepts provides facilities for creating components that encapsulate behavior 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 a heterogeneous 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 modeled and successively refined towards an implementation.

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

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

Other approaches use a number of domain specific formalisms with which different aspects of a system's behavior 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.

Unlike these approaches that use special languages to represent other formalisms this approach uses the same object-oriented time Petri net kernel language. This is advantageous, as only a single modeling language is involved in modeling both a system and a formalism. In addition the semantics of combining multiple formalisms are completely given by the 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.

In the second part of this paper, these concepts were applied to a real world modeling task in an industrial setting. The graphical and intuitive nature of the modeling method improved communication in the interdisciplinary team and proved to be a significant step forward to integrating the knowledge of the domain experts into a common model. This led to a high degree of acceptance of this modeling technique.

The Petri net based formalism allowed a finer grained mapping of certain dynamic properties of the system than the purely statistical calculations used before, especially when complex interdependencies between successive cycles were involved - not a rare occurrence in heterogeneous systems. The resulting estimates for the throughput of the modeled system were as a result more accurate. In addition, the tool's animation facilities made it possible to view the dynamic behavior as it evolved thus facilitating the understanding of certain aspects and interdependencies of the system's dynamics.

6 Bibliography

[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, LNCS[1] 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

1. Lecture Notes in Computer Science

[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 Dissertation ETH Zurich 1996, VDF Verlag Zurich, December 1996.

[9] Esser R., An Object Oriented Petri Net Language for Embedded System Design, Proc. of the 8th International Workshop on Software Technology and Engineering Practice (STEP), London, England, July 1997

[10] Fehling R., A Concept of Hierarchical Petri Nets with Building Blocks, In LNCS 674, Advances in Petri Nets 1993, Springer-Verlag, 1993

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

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

[13] 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

[14] 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

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

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

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

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

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

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

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

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

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

[24] Stalder, R: Untersuchungen zur Simulation von UPH-Raten für die Epoxy Diebonder, 20. 01. 1997, Internal Memo, ESEC SA, Cham, Switzerland

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

[26] 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