CodeSign: An Embedded System Design Environment

Robert Esser Jürgen Teich
Lothar Thiele
Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH), CH-8092 Zürich
Switzerland
email: thiele@tik.ee.ethz.ch

December 16, 1997

Abstract

In this paper a modelling environment is described for the automated design of embedded systems. The basic model of computation 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 model constitutes a major part of the CodeSign design methodology [8,31] developed at the ETH.

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. 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, therefore it is desirable to support a number of formalisms that are well suited to modelling particular parts of an embedded system.

The methodology 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 can also be used for efficient code generation and system analysis. 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.

2 Basic Model of Computation

The basic model of computation is a combination of object oriented concepts and Petri nets. The Petri net class used is a composition of high level Petri nets [18] and time Petri nets [16]. Time plays an important role in embedded real time systems and any language that purports to model such systems must include a concept of time. Time models for Petri nets and their analysis are described in [3]. Two basic Petri net models for handling time have been proposed - time Petri nets [16] and timed Petri nets [22]. Timed Petri nets and the subsequently developed stochastic Petri nets have been mainly used for performance evaluation.

Time Petri nets associate a finite enabling duration with each transition in a net. For example, in [32], real-time systems have been modeled with time Petri nets. They are adequate for expressing temporal constraints. Analysis techniques for formal verification of time dependent behaviour 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.

Example 1 *Figure 1 is an example of high level time Petri nets used as the basic model of computation. This class of Petri nets is an extension of 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. Transitions are inscribed with tuples* (α, β) *representing time constraints, guard functions* (g) *that define additional enabling conditions and functions* (f) *to determine the type and values of tokens produced as a result of a transition occurrence.*

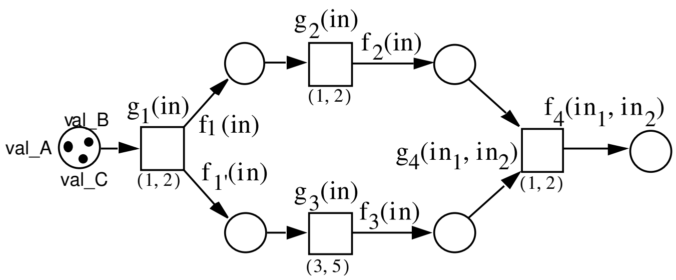

Figure 1: A high level time Petri net where tokens represent data and transitions contain timing intervals and functions

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 [12], [20], [32].

Recently there have been attempts to combine the structuring techniques of object oriented programming with Petri nets [14], [24]. 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. For example, a filter can be described as a data flow graph transforming integer data to integer data. A refinement may involve defining the number of bits used to represent this data and how an overflow is handled (truncation, rounding or exception), see also Figure 2.

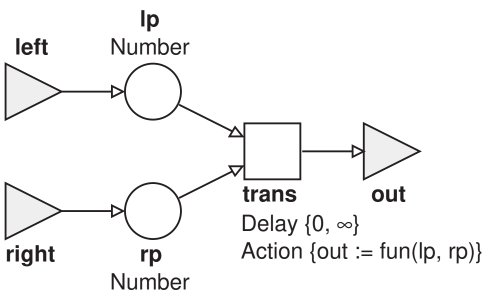

Figure 2: Showing the abstract superclass of classes Adder and Multiplier. It defines the behaviour and state with the above Petri net and also an abstract function fun. Interfaces are represented as triangle shaped elements.

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.

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.

Example 2 *Figure 3 shows a component that implements the function* $f(a,b) = \mathrm{res} = 2a + b^2$ *and contains three embedded components using the object oriented time Petri net (OOTPN) kernel language.*

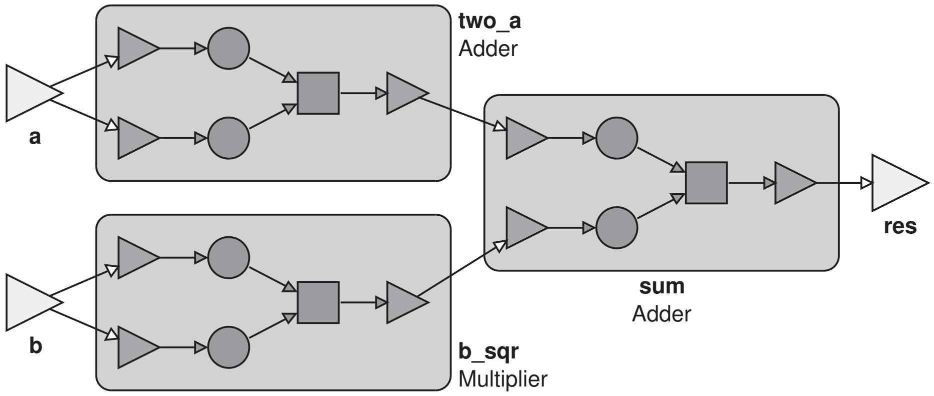

Figure 3: Graphical representation of a component containing embedded components.

3 Support for Other Formalisms

A design methodology should allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled. These models may be used within a design process with the goal of describing embedded systems. The large number of modelling languages available is an indication that the concept of one general model is unrealistic, hence the necessity for heterogeneous specification models. Examples for reactive system descriptions are finite state machines, behaviour state machines [27], codesign finite state machines [28], StateCharts [11], program state machines [9], CSP, Esterel and SDL. For transformative systems, the SDF model [15] is the most well known. Other models which mix models of computation are Boolean Data Flow, hierarchical flow graphs [2], extended syntax graphs [1], Petri nets and programming languages like C, Occam or VHDL.

One approach for supporting multiple formalisms is with a simulation backplane, where individual tools supporting different formalisms communicate during simulation. This has the advantage that only the communication with the simulation backplane is implemented and not the individual tools themselves.

However different tools support various methods of refinement and have diverse user interfaces. Moreover, there is no possibility to formally analyse the whole system, e.g. proving that the system correctly implements functional and timing requirements.

Alternatively an environment can be defined where different formalisms are described in an implementation language and communicate with each other through defined interfaces. This approach has the disadvantage that $n^2 - n$ interfaces are required to support n formalisms and that no high level basis exists for these interfaces which allows models described with multiple formalisms to be analysed.

The approach described in this paper can be characterised as follows:

- It is based on a formal object oriented kernel language.
- Graph grammars [19], [21] and corresponding 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.

In comparison to other approaches, this new concept has the following advantages:

- It is possible to capitalise on formalism specific tools and methods, for example for efficient code generation, formal verification and simulation.
- 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.

3.1 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. 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 of a formalism.
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.

4. Functions are then written representing the productions defined by the SemGG and are associated with a particular class in the object oriented environment.

5. To complete the description of a formalism functions are created that define how an element of a formalism is graphically represented (shape, weight, state, etc.).

A formalism can be contained within a component representing the subsystem model. This component has kernel interfaces through which the formalism can communicate with other parts of the model described by the kernel language or other formalisms. The class that defines the component can also define the editing facilities (productions) that are appropriate for the formalism.

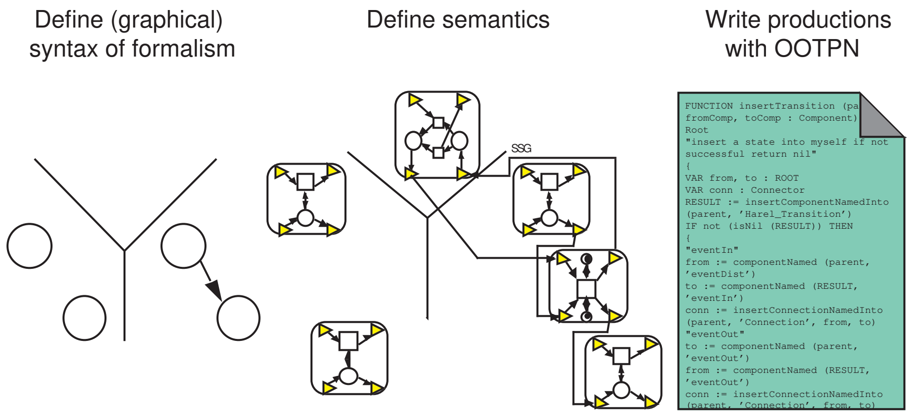

Figure 4: Syntax, Semantic and implementation Graph Grammar productions describing the addition of a state-transition in a simple Finite State Machine formalism.

The approach taken differs from other well known approaches, where for each new formalism, a new data structure and corresponding language is introduced. The relation between this introduced data structure and the underlying kernel language is defined using transformations. The same is true for the relationship between the introduced data structure and the graphical representation.

In the CodeSign environment, the semantics and graphical representation of a formalism is defined in terms of graph grammar productions and Petri nets. *Consequently, no data structures are introduced.* The act of creating a model in a supported formalism creates an object oriented Petri net which is visualised as the original formalism, hence no translation takes place. The structure of a subsystem modelled in a supported formalism is present and can be used for optimal formalism specific code generation, analysis etc. Leaf nodes of the instantiation hierarchy consist only of Petri net elements. As a result, the

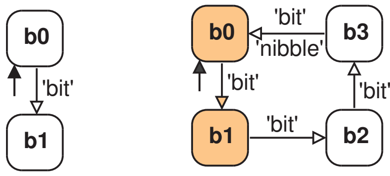

Figure 5: The Class Bit_2 implementing a 2 bit counter (left hand side), the Class Bit_4 implementing a 4 bit counter (right hand side)

following advantages in comparison to the use of an introduced data structure (and language) are obtained:

- The extension of the design environment with a new formalism can be made by the user as no change to internal data structures is necessary. Only the graph grammar productions for the (graphical) syntax and the (Petri Net) semantics must be defined.

- Properties of the kernel language are *automatically* inherited by the new formalism. This includes, for example the hierarchical structure (component) concept, object orientation, the capabilities for simulation and formal analysis, in addition to the time model.

3.2 Other Formalisms in an Object Oriented Environment

As described previously, other formalisms are defined with the kernel language. As with the components of the kernel language, a subclass may be extended and refined in a number of ways. Assuming that the functions representing SemGG productions allow a choice of component classes to be instantiated, subclassing enables abstraction and refinement to be used in all formalisms.

Embedding is a formalism specific feature and usually represents hierarchical structuring. Formalisms that do not support hierarchy, such as Synchronous Data Flow Graphs, e.g. [15], cannot make use of embedding at the formalism level. In contrast StateCharts [11] do support hierarchy and can benefit from the advantages of code sharing.

Example 3 *The following figures show an example of subclassing and reuse using the StateCharts formalism which has been 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 5 shows a simple two bit counter (left hand side). Figure 5 (right hand side) extends the 2 bit counter to 4 bits. Bit_4 is a subclass of Bit_2 and has been enhanced with the addition of two states (b2, b3) and 3 state-transitions.* The inherited states and transitions are shown as being shaded. *Finally in Figure 6 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.*

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 in different formalisms. Class Bit_4 reuses the states and state-transitions defined in class Bit_2.*

Class Bit_8 consists of two instances of Bit_4. As the two component hierarchical states are instances of the same class they only differ in their dynamic state information.

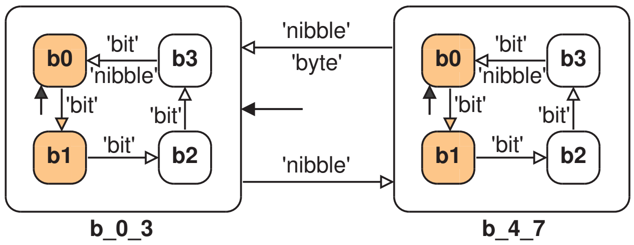

Figure 6: The Class Bit_8 implementing a 8 bit counter

3.3 Communication between components modelled using different formalisms

To illustrate the use of multiple formalisms consider Figure 7. This figure shows a model of a multiplexer and demultiplexer where the boolean data flow (BDF) formalism [29] (switch, select and data flow nodes) is coupled with a finite state machine (FSM). The function of this system is to alternatively apply the functions Op_1 and Op_2 to an incoming stream of data. Here it can be seen that the FSM controls the operation of the switch and select nodes and hence the flow of data tokens through nodes Op_1 and Op_2.

Figure 8 shows the internal object oriented Petri net representation of the individual formalism specific elements. The dynamic (state) and static graphical representation as well as the semantics in terms of Petri Nets is defined by Graph Grammar productions as described in Section 3.1.

Due to the simplicity of BDF and the simple FSM class chosen there is a one to one correspondence between formalism components and their underlying Petri net representation. In more complex formalisms, e.g. Harel StateCharts, implicit formalism specific mechanisms, such as, in the case of StateCharts, event distribution and collection, would also need to be defined in terms of Petri nets.

Here it also can be seen how communication between the two different formalisms is implemented. In general the underlying Petri net representation defines the communication abilities of a particular formalism. In this example FSM events are interpreted as data flow tokens. This mechanism also allows two or more state machines to be coupled together, thus allowing asynchronously coupled state machines to be modelled.

Figure 9 shows the equivalent Petri net representation of the same functionality as described in Figure 7. This representation is obtained after flattening

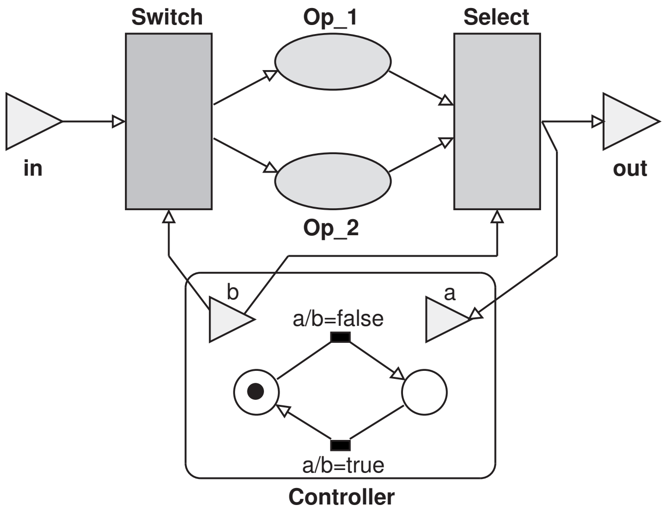

Figure 7: A model of a multiplexer/demultiplexer system using the boolean data flow (BDF) and finite state machine (FSM) formalisms.

the component and class hierarchies. It can be seen that specialised formalisms are easier to understand assuming the semantics of the formalisms are well defined. Note that token *carry* objects, as we are dealing with high level nets. In Fig. 9, the tokens controlling the operations of the switch carry the value *true* with enables just one of the two paths. Moreover, in order to model finite execution and communication times, e.g. of the controller, the operations, or the switch, it is possible to assign timing intervals to the corresponding transitions. As a result, a transition fires if it was continously enabled for some latency τ which is contained in the timing interval.

4 Design Methodology and Design System

A design methodology defines the process of designing a system, typically from conception to implementation. It may require the use of different models at various stages to highlight particular system characteristics that need to be addressed. What follows is a small selection of design systems on which the proposed approach is based on: COSYMA [1], Olympus [2], Ptolemy [6], Model-based Object Oriented System Engineering (MOOSE) [7], Structured Analysis [10], structured development for real-time systems [13] and Real-time Object Oriented Modelling (ROOM) [17]. The FOCUS method [33] for the design of distributed embedded systems is based on the processing of data streams and correspondingly, also uses a different time model than the present approach.

An object oriented methodology describes how systems are designed, based on the concept of communicating objects. The choice of modelling language is the main difference between various methodologies. ROOM [17] uses hierarchical finite state machines and OMT [23] uses hierarchical finite state and data flow representations of an objects behaviour. MOOSE [7] uses object interaction

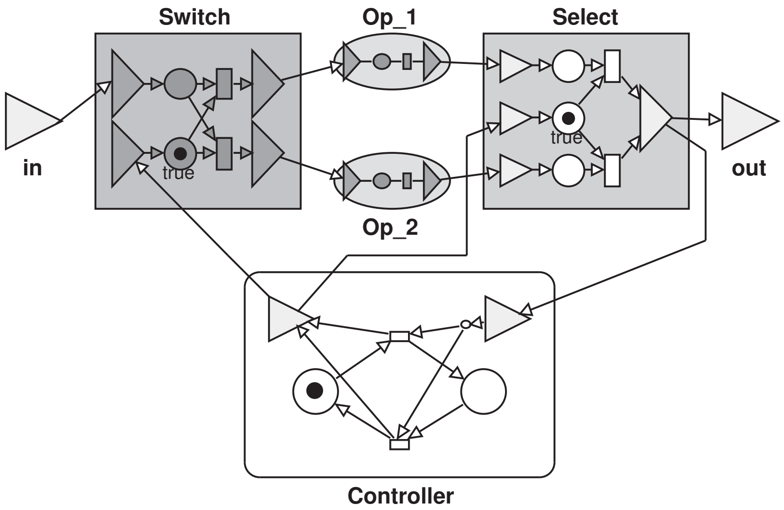

Figure 8: The internal Petri net representation of all elements used to model the multiplexer/demultiplexer system.

diagrams and the Booch method [4] simply defines relationships between objects. The following proposed methodology combines aspects of ROOM, OMT and the model based codesign methodology [26] and includes the following steps:

1. Define the system and its boundary with the environment.
2. Formulate scenarios that the system must implement.
3. Using object oriented high level Petri Nets and/or a derived formalism create a sufficiently abstract model of the system's environment.
4. Define a set of kernel language components representing the system structure.
5. Key components are developed and prototyped and the system is validated at various stages during the development of the component class definitions.
6. Partition the system model, introduce interfaces between implementation devices and analyse the result.
7. Refine the model by repeating the previous two steps in such a way that backtracking is possible. Generate code as input into software and hardware implementation processes.

Steps 3 to 7 are directly supported by the CodeSign framework. Figure 11 shows the environment and the tools of this framework. Given a specification, the tool for design space exploration obtains performance and cost parameters of system components from VHDL and C module libraries, exploits architecture

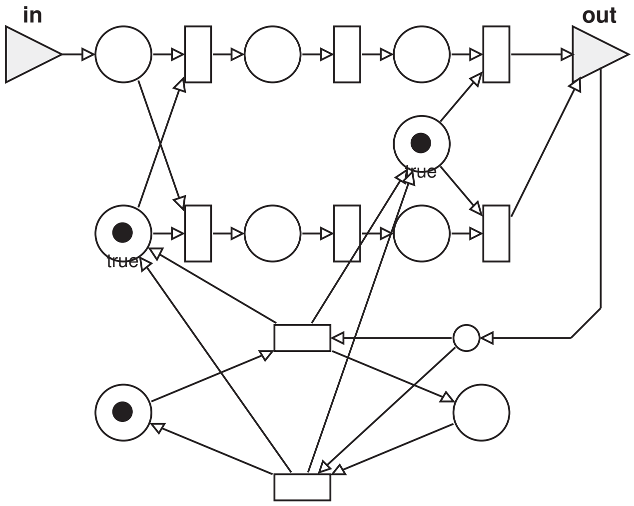

Figure 9: The equivalent Petri net representation of the multiplexer/demultiplexer system.

tradeoffs. Currently, the design space exploration tool [25,30] is based on genetic algorithm optimisation. The computational model of the initial algorithm specification is a signal flow graph. The class of architectures and their relations between communication and operations defined in the signal flow graph are captured in a specification graph, for details see e.g. [25,30]. The parameters are captured by constraint functions in the corresponding component classes, see Figure 10.

```
CLASS RISC.0 : Hardware_Node.0 DIRECTORY './Projects/Video/'
FUNCTION partitioningInformation : Collection
"return a collection of tuples with name and value"
{
RESULT := [['RESOURCE_COST', 100]]
}


CLASS BM.0 : Video_DFG_2_1.0 DIRECTORY './Projects/Video/'
FUNCTION implementationTargets : Collection
"a collection of class component names and execution times"
{
RESULT := [['BMM', 22], ['DSP', 60], ['RISC', 88]]
}
```

Figure 10: Constraints functions that define the possible target classes and time delay required to implement a component on the particular target and the target implementation cost.

Creating new design points, or design library modules for specified subnets, is possible by clustering subnets and synthesizing a VHDL-design (**`nettovhdl`**), a C-module (**`nettoC`**) including a HW/SW-interface (**`intgen`**), or a mixed HW/SW implementation for the given architectural constraints. For example, the module **`nettovdhl`** works tightly with the Synopsys Behavioural and Logic Synthesis tools. It generates a VHDL-entity which implements the subnet as well as all necessary script files for synthesizing not yet implemented node functionality using the behavioural compiler. It also generates the communication ports required for connecting with synthesized behavioural compiler components. In addition stimuli files are automatically created for simulating the synthesized code.

`nettoC` generates a task-level description of a subcomponent in C together with a simple scheduler for these tasks. The primitives used to communicate between these components are based on a subset of PVM.

Depending on the chosen architecture, `intgen` (see also [34]) generates interfaces in VHDL (I/O-control, MMU, etc.) and C (driver, I/O-commands, etc.). From this point on, target-dependent synthesis takes place including technology mapping in VHDL, e.g. lca-files for FPGA prototyping using Synopsys logic synthesis and XilinX Xact software, and target-dependent code generation (e.g. assembler code generation and interface refinement (driver selection, scheduler implementation)).

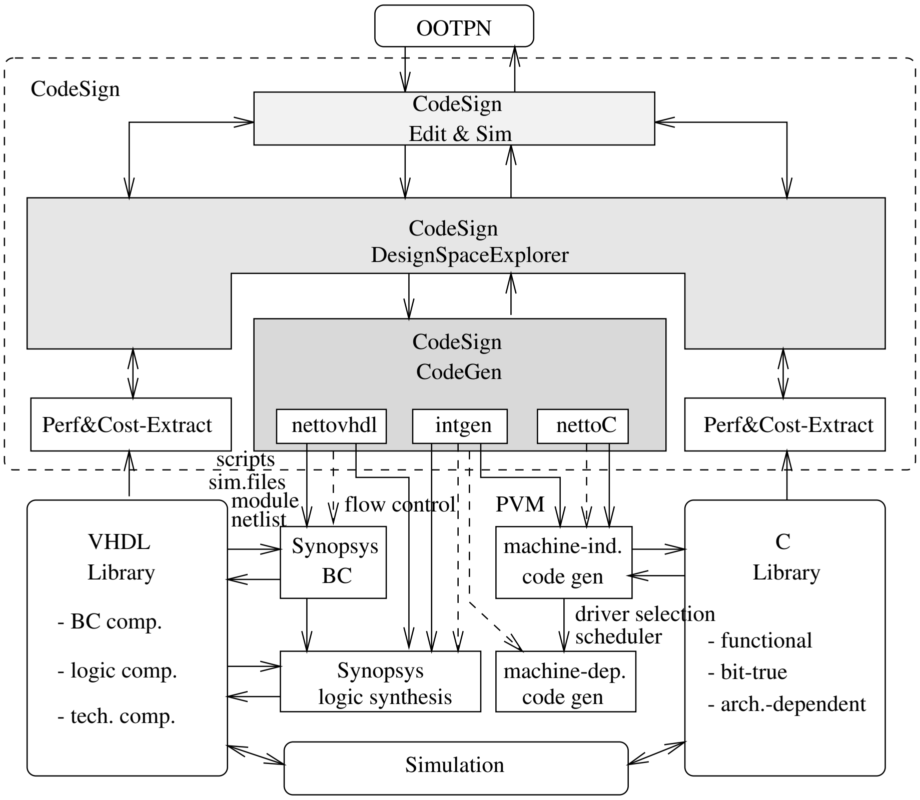

Figure 11: The CodeSign Framework

5 Examples

The following section contains two examples which serve to clarify the design methodology and how different support tools are embedded into the framework.

5.1 Telephone

The example of a telephone will not show those aspects of the traditional codesign approach, i.e. partitioning, interface generation, hardware and software synthesis etc. Instead, emphasis will be placed on the highest level of abstraction - system specification and modelling. Because of space restrictions, only some intermediate representations can be shown. In particular, Figure 12 shows the model of an exchange line card, mixing hierarchical state machine specifications and Petri nets.

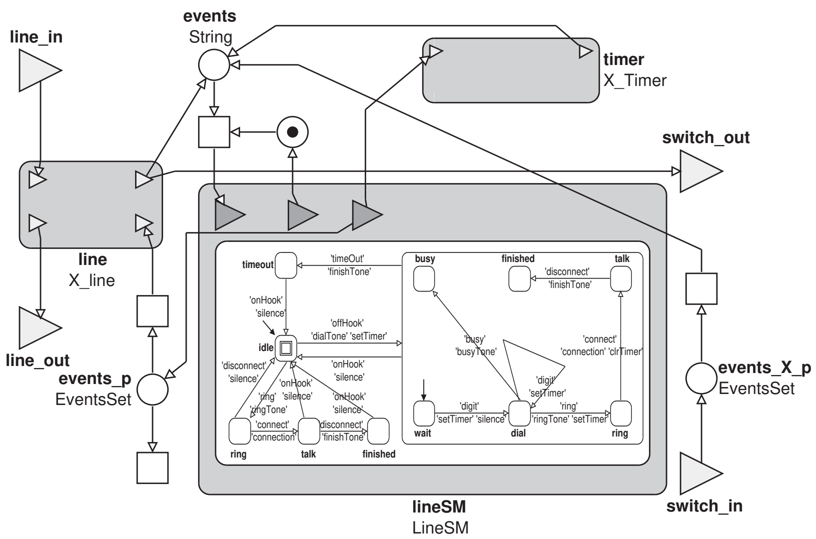

Figure 12: Exchange Line Card

Figure 13 shows a version of the dialer class modelling both tone and pulse dialing. Here, different formalisms are mixed, i.e. synchronous data flow graphs (top component) and Petri nets (bottom component).

The user interface of the CodeSign framework is shown in Figure 14. In particular, the list views on the left hand side represent and manipulate the class and instantiation hierarchies. The class window shows a component which contains two further components. One of which uses a synchronous data flow (SDF) model of computation (DTMF encoder), the other one using Petri net semantics. The overlayed component window shows a controller with a model of computation based on Harel's StateCharts formalism (dial controller).

Main aspects of the whole design trajectory is the support for stepwise refinement using the described object oriented techniques and a powerful configuration mechanism. This mechanism enables the exchange of components for refinement, simulation, verification and implementation purposes.

5.2 Video Codec

The next example shows how the presented approach is applied to hardware design. Figure 15 shows a block diagram of a video codec used for the compres-

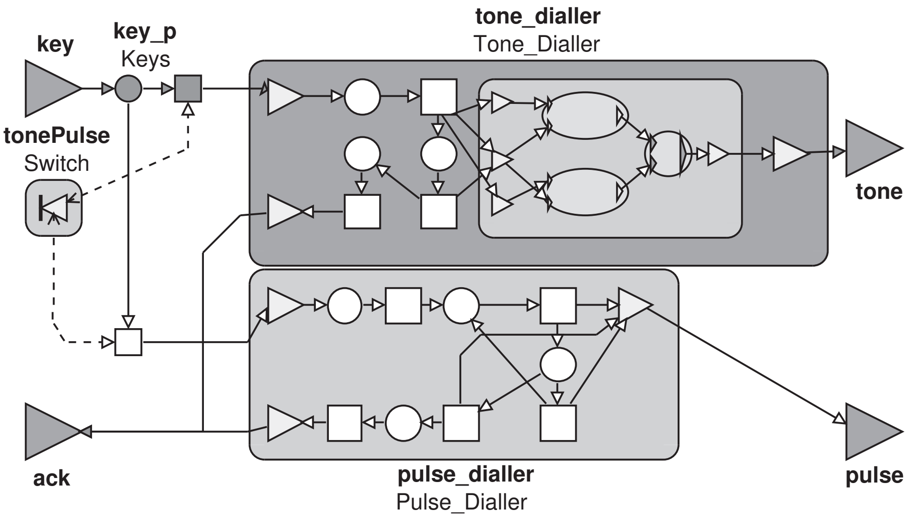

Figure 13: Class modelling both tone and pulse dialing

sion of video signals. This behavioural specification has been modelled using the synchronous data flow formalism and is shown in Figure 16.

In this model communication is explicitly modelled with communication nodes as this allows the communication between nodes to be parameterised. The individual components are instances of classes which within the object oriented framework implies that behaviour may be inherited, refinement of components is well supported and that many different configurations can be supported within a single model. This model can also be inserted within a test environment and simulated with or without animation.

Figure 17 shows the architectural template describing a system with (possibly) three shared buses (SBS, SBM, and SBF for slow, medium, and fast), each having a different bandwidth, two memory modules (dual (DPFM) and single ported (DM)) two programmable RISC processors (RISC1 and RISC2), a signal processor (DSP), several specific hardware modules (block matching (BMM) and DCT/IDCT operations (DCTM)), a subtract/add module (SAM), a Huffman coder (HC) and I/O devices (INM) and (OUTM). This template describes a wide range of possible implementations. The connections between components describe whether the communication is uni or bidirectional. It can also be seen that a local bus exists between the DPFM and the BMM module whose characteristics are embodied in the point to point (PTP) component. Depending on system constraints a subset of the components will be chosen that fulfills the required system constraints. Figure 18 shows how these components and their interconnetions are represented in CodeSign.

Figure 10 shows two typical functions, the first is associated with all classes whose components are nodes in the architectural template and describes tuples of constraint names and values. The second function is associated with all classes whose components are elements of the problem graph. This function returns tuples consisting of architectural template class names and the execution

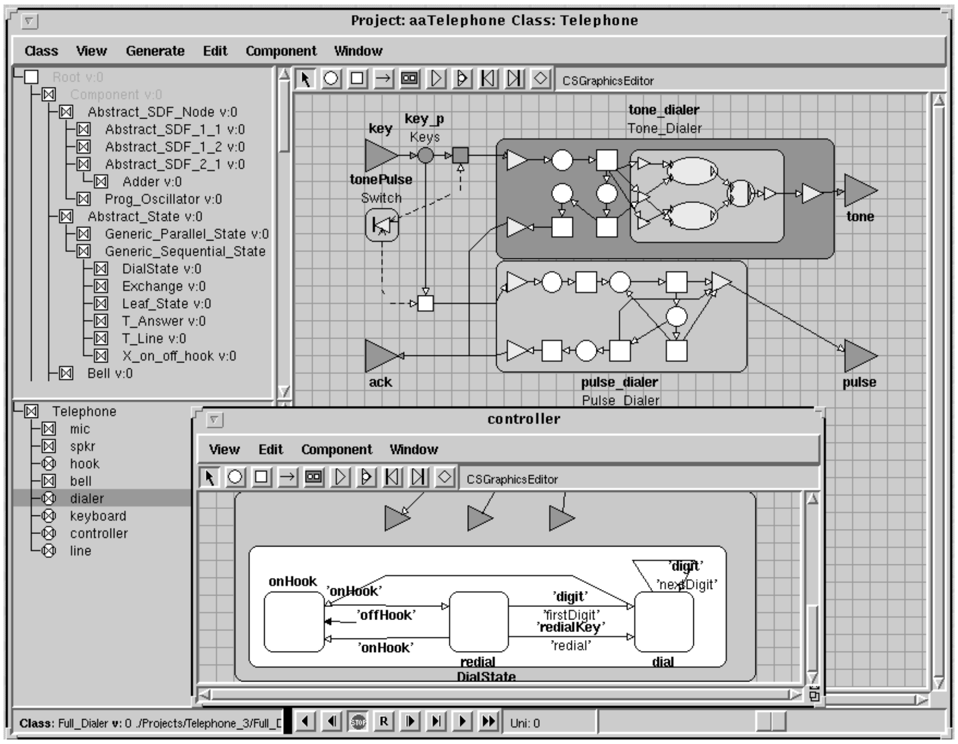

Figure 14: The CodeSign User Interface

time estimate of the particular component on an instance of an architectural class. Therefore it can be seen that the RESOURCE_COST of a RISC processor is 100 (dollars, amperes, square meters....) and that the block matching (BM) component can be implemented on three types of architectural components with different execution times.

Once the problem graph has been validated and the architectural template defined data is generated for use by the tool DesignSpaceExplorer [25,30], see Fig. 11. This tool uses an Evolutionary Algorithm for exploring the design space of allocations α (instantiations of modules in the implementation), bindings β (binding of nodes in the problem graph to instantiated modules in the target architecture), and scheduling τ (assignment of start times to processes). For each individual of a population of possible implementations, α and β are coded and optimised within the Evolutionary Algorithm (see Fig. 19) whereas an external scheduler determines an optimized schedule for a given allocation and binding. Using these informations, a fitness function (see [30] for useful fitness functions) is evaluated. Based on these fitness values, the Evolutionary Algorithm generates new populations, and so on until a certain number of iterations has been explored or until individuals in the actual population have reached a certain threshold fitness.

In Fig. 21 (right), the explored design points are shown including the highlighted Pareto-points: each of these points has the property that no other point in the design space dominates it in all design criteria (in the case of the video

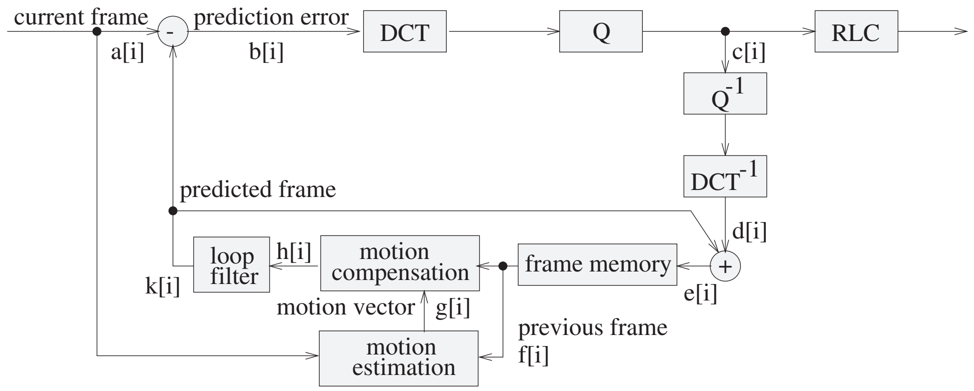

Figure 15: Behavioural specification of a video codec for video compression

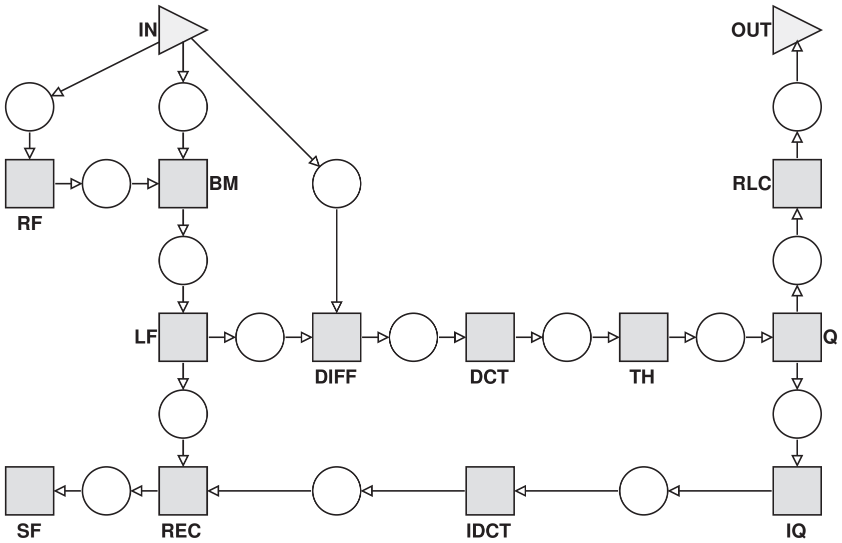

Figure 16: Problem graph of the video coder where function nodes are represented as squares, communication nodes as circles and interfaces as triangles.

codec: the two dimensions cost and frame period are to be minimised). Hence, all Pareto-points are optimal in this sense. The tool can also display the properties of each design point in a Gantt-chart (bottom) including the allocation of resources, the binding of functionality to resources and a schedule. Different optimisation goals (e.g. Pareto-optimisation, maximal resource usage, minimal cost, etc.) may be selected. In Fig. 20 a Ganttchart for one Pareto-individual is shown as well as the allocation of this implementation. The implementation uses the modules FM, BMM, INM, SBF, SAM, DCTM, HC, and OUTM. With a period of $P = 22$ cycles, this implementation was one of the fastest implementations found in this example.

Finally, for each design point, a refined model of the system may be created

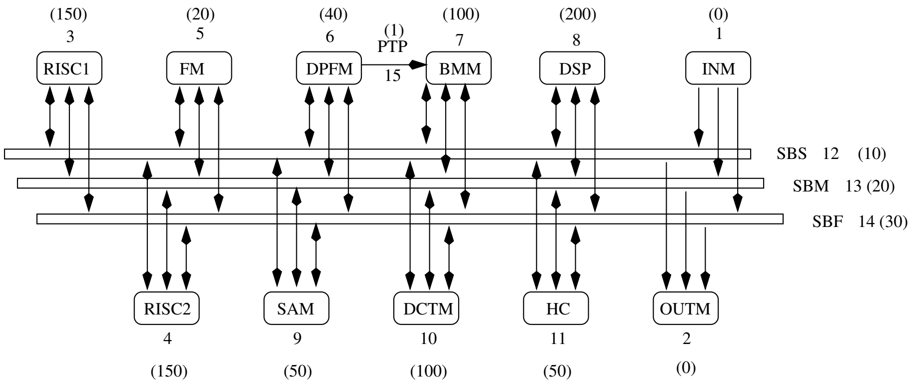

Figure 17: Architecture template of architectures to implement the problem graph of the video codec. The values in brackets represent the basic cost of each resource in case it is instantiated (used) in an implementation.

where functional nodes are components of architectural devices. From this description interfaces can be inserted, either being instantiated from a library or generated automatically [34]. The resulting system can then be further simulated and validated.

6 Conclusions

This paper presents a new approach to the design of embedded systems. This approach is based on time Petri nets augmented with object oriented concepts.

Other approaches often neglect the question of the semantics of the combination of formalisms within one specification. Here, a single modelling language is involved in modelling 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.

References

[1] Benner Th., Ernst R., Könenkamp I., Schüler P., Schaub H.-C., A Prototyping System for Verification and Evaluation in Hardware-Software Co-synthesis. Sixth IEEE International Workshop on Rapid System Prototyping, 1995.

[2] De Micheli G, Ku D., Mailhot F., Truong T., The Olympus Synthesis System, IEEE Design and Test of Computers, 1990.

[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.

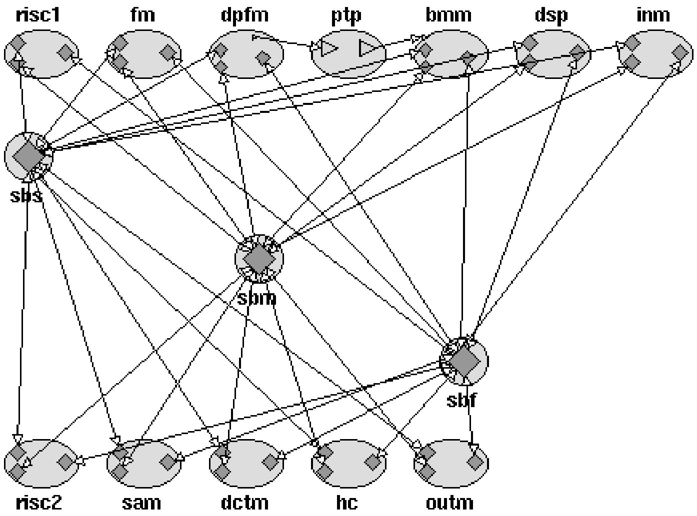

Figure 18: Architectural template of architectures used to implement the video coder in CodeSign

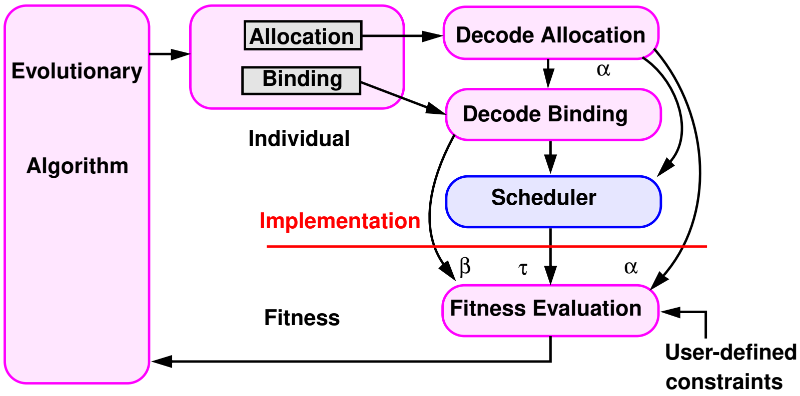

Figure 19: The decoding of an individual to an implementation.

[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] Kalavade A., Lee E.A., A Hardware Software Codesign Methodology for DSP Applications, IEEE Design and Test of Computers, 1993.

[7] Green P., Rushton P., Beggs R., An Example of Applying the Codesign Method Moose, Proceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994.

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

[9] Vahid F., Narayan S., Gajski D., SpecCharts: A VHDL Front-End for Embedded Systems, IEEE Transactions on CAD, 1995.

[10] Yourdon E., Constantine L.L., Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice Hall, 1979.

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

[12] Jensen K., Colored 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] Ward P.T., Mellor S.J., Structured Development for Real-Time Systems. Prentice Hall, 1985.

[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] Selic, Gullekson, Ward, Real-Time Object-Oriented Modeling, Wiley 1994.

[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] Rumbaugh J., Blaha M., Prmerlani W., Eddy F., Lorensen W., Object Oriented Modeling and Design, Prentice Hall, 1991.

[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.

[25] Blickle T., Theory of Evolutionary Algorithms and Applications to System Synthesis, PhD thesis, ETH Zurich, 1996, VDF Verlag, 1997.

[26] Buchenrieder K., Rozenblit J.W., Codesign: An Overview, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press, 1995.

[27] Takach W., Wolf A., An Automation Model for Scheduling Constraints in Synchronous Machines, IEEE Transactions on Computers, 44, 1-12, Jan. 1995.

[28] Chiodo M., Guisto P., Hsieh H., Jurecska A., Lavagno L., Sangiovanni-Vincentelli A., A Formal Methodology for Hardware/Software Codesign of Embedded Systems. IEEE Micro, August 1994.

[29] Buck J. T., Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., UCB/ERL 93/69, 1993.

[30] Teich, J., Blickle, T., Thiele, L., System-level Synthesis using Evolutionary Algorithms. J. Design Automation for Embedded Systems, 3(1), pp. 23-58, 1998.

[31] Esser, R., An Object Oriented Language for Embedded System Design. 8th Int. Workshop on Software Technology and Engineering Practice, London, July 1997.

[32] Sen, G., Brummer J., An Approach to graphical specification of real-time systems. Proc. First Int. Conf. on Software Testing, Reliability and Quality Assurance, pp. 95-99, New Delhi, India, December 1994.

[33] Broy, M., Dederich, F., Dendorfer, C., et al. The Design of Distributed Systems - An Introduction to FOCUS, Techn. Report No. TUM-I9202, TU Munich, 1992.

[34] Eisenring, M., Teich, J., Thiele, L., Rapid prototyping of dataflow programs on hardware/software architectures. Proc. of HICSS31 - the 31st Hawai'i Int. Conf. on Syst. Sci., Kona, Hawaii, January 1998.

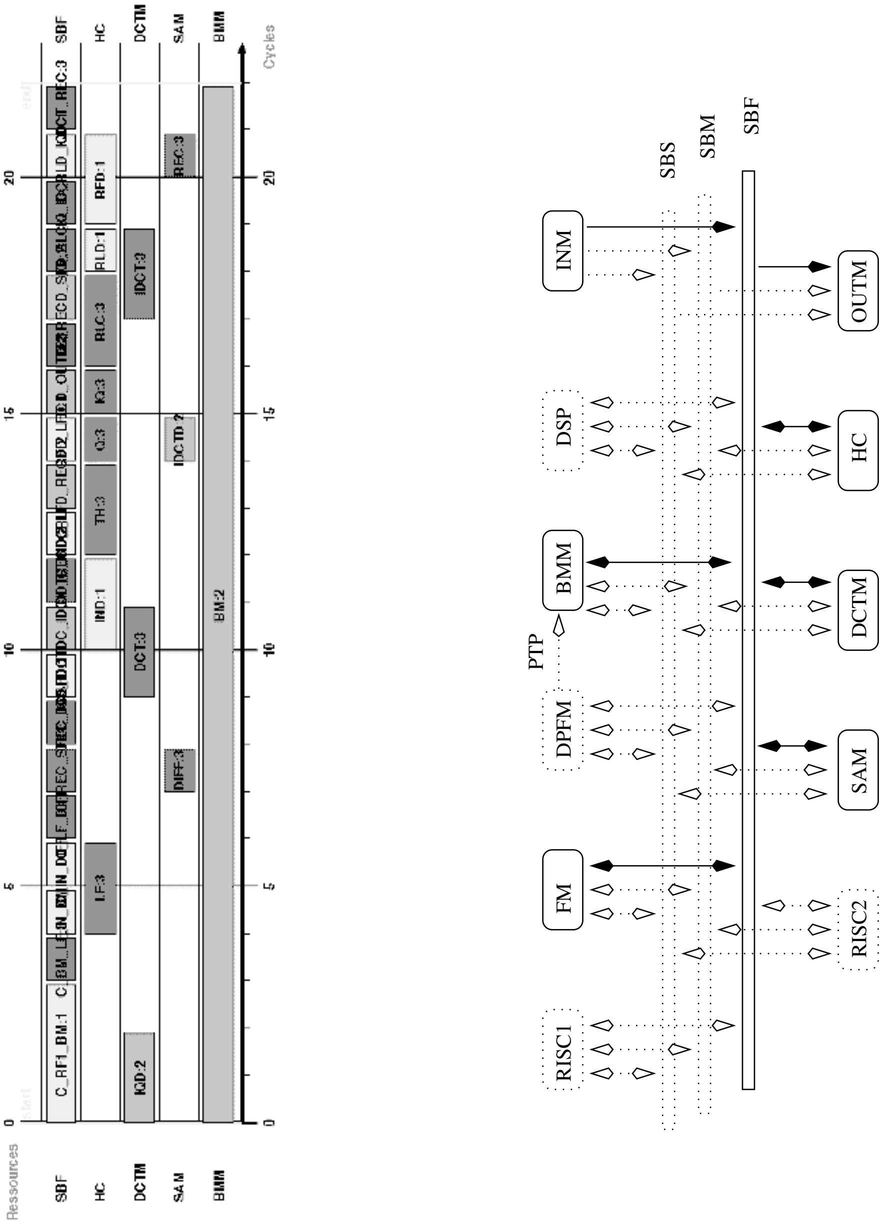

Figure 20: Ganttchart and architecture of the fastest implementation (cost $c = 350$, period $P = 22$).

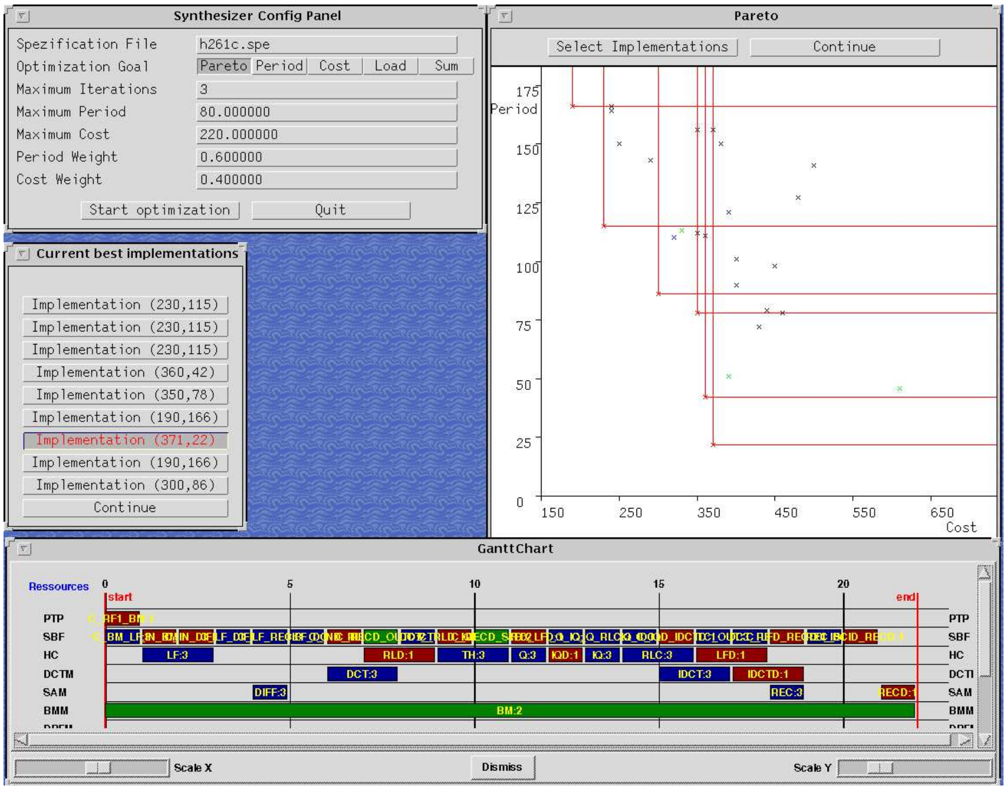

Figure 21: Tool for design space exploration in CodeSign.