System Level Design

Franz J. Rammig

Universität-GH-Paderborn
FB Mathematik/Informatik
D – 4790 Paderborn
Tel.: 49.5251.60.2069
Fax.: 49.5251.60.3427
e-mail: franz@uni-paderborn.de

Abstract:

In this contribution, some aspects of system level design are discussed. After a short characterization of the term system level, most emphasis is laid on modelling aspects. Various modelling techniques are discussed and a specific model, extended Pr/T-Nets are introduced in detail. A short overview of other system level activities concludes this paper.

1. Introduction

For a long time, different engineering disciplines have been in existence somewhat independently. As a consequence, computer support has been developed individually. In the case of electronics design there has been an evolution from tools at low levels of abstraction to more and more abstract levels. This trend now reaches a level, where electronic components can be treated no longer independently from their context. This is the challenge this paper deals with. As the basic problem to be solved is to understand it and understanding something means to construct a model of it, most emphasis will be laid on modelliing. The paper therefore is organized as follows: In sections 2, 3 and 4 an idea about the system level and related design activities will be given. After an overview on various modelling techniques (section 5) a specific modelling technique (extended Pr/T-Nets) will be discussed in section 6. In section 7 a discussion of system level design activities shall be initiated. Section 8 is devoted to show communalities between system level design and concurrent engineering.

2. System Level: The Electronics Engineer's Point of View

Today in the area of digital hardware design there is a fairly widely accepted scheme of 6 abstraction levels [RA89]. This scheme is orthogonal to the different views a system is looked at. As example for this Gajski [GA87] identifies three different views: behaviour, structure, and geometry. Additional views are possible, e.g. a test view [RA89]. Especially for VHDL other organizations may be suited better [EH92]. However, for the purpose of this discussion, Gajski's scheme is sufficient. At higher levels the behaviour view is of main interest. So in this context the levels of abstraction are discussed mostly with this view in mind. In order to make the

J. P. Mermet (ed.), Fundamentals and Standards in Hardware Description Languages, 109–151.

concept of abstraction and by this the system level more visible, a bottom up presentation has been chosen.

The lowest level (level 1) is usually called the *electrical level*. Here it is modelled, how electrical circuits built from resistors, capacitors, etc. behave over the time axis. This is done by a system of differential equations. i.e., both the time axis and the observable values are represented by a continuous domain. It should be noted that the geometrical view of this level is the (metric) layout which does not constitute its own level.

Electric Level:

- Modelling concept: differential equations
- Timing model: continuous real time
- Observable values: continuous reals

The *switch level* (level 2) is the next abstract one. This level is fairly well accepted in digital MOS design but makes sense in other digital designs as well. The abstraction comes from modeling transistors as ideal on/off switches and the connections in between as discrete capacitances. So the value domain is a discrete one where a value is given by a pair consisting of a logical interpretation and a strength, both within a finite domain. This abstraction introduces uncertain values. They are handled either by introducing additional "values" or by representing uncertainty by enumerating all possible values [LR83]. The time domain may still be a continuous one. Other approaches like MOSSIM [BR81] have a discrete time in mind (unit delay assumption). This leads to a concept to model switch level circuits by finite automata.

Switch Level:

- Modelling concept: discrete equations
- Timing model: continuous real time
- Observable values: pairs (value, strength)

The *gate level* (level 3) has a long tradition in digital system design. It has a very nice mathematical background in Boolean algebra. However, this models only the timeless behaviour. So some additional concepts have to be considered in order to cover the time axis as well. Various models of inertial delay and (inertia-less) transport delays have been discussed in the literature [KA90,Ra80a,Ra92] and efficient implementations of such models in simulators have been achieved. In the ideal case the value domain is restricted to Boolean values 0 and 1 while the time domain remains continuous. Again, the problem of uncertain values forces one to introduce additional values. By this in most cases the underlying algebra is no longer a Boolean one. Even the concept of different strengths is carried over from the switch level in some cases. The operators, however, are always Boolean (logical) operators. This finally constitutes this level. If the value domain is restricted to 0,1 and the time model is unit delay then the modeling concept for this level is exactly a system of Boolean equations.

Gate Level:

- Modelling concept: "Boolean" equations
- Timing model: continuous real time
- Observable values: "bits" (may be multivalued)

Further abstraction comes with the *register transfer level* (level 4). At this level a specific mode of operation is assumed. There are components that continuously observe their specific conditions. Whenever the condition of a component becomes true this component performs its specific operation. In any case such an operation may be interpreted as a transfer of data between registers where the data may be modified during the transfer. It is this point of view that gives this level its name. Abstraction here originates from implicitly underlying a specific mode of operation. In addition the elementary components used at this level are more complex (e.g. registers, ALUs, etc.) abstracting from their implementation. The value domain at this level is given by (uninterpreted) bitstrings while the timing model is usually the counting of clock ticks. So the time domain has now become a discrete one as well. The register transfer level is very helpful in clean synchronous designs, it forces one to somehow design in this manner. This level has been studied intensively in academic institutions but is much less popular in industry. As a consequence, nearly all of the numerous register transfer simulation systems (e.g. CASSANDRE [ME70], CDL [CH79], DDL [DD75], KARL [HA77]) are more often used in universities only.

Register Transfer Level:

- Modelling concept: Guarded commands
- Timing model: discrete real time (clock ticks)
- Observable values: bitstrings

At the *algorithmic level* (level 5) the reactive point of view at the register transfer level is inverted to an imperative one. While at the register transfer level the system is looked at from the eyes of the individual components, at the algorithmic level the controller's point of view is taken. In contrast to ordinary algorithmic descriptions, however, concurrency plays an important role in hardware design and therefore also at this level of abstraction. Therefore highly concurrent algorithms are usually described. While at the register transfer level it is specified precisely what conditions cause operations to be carried out, it is abstracted from this information at the algorithmic level. Only the logical point of time, when an operation has to be carried out is identified. All the remaining details are hidden away by assuming the imperative mode of operation of a system. The domain of values may be freely definable but is usually restricted to bitstrings with interpretations attached. The timing model is either still a counting of clock ticks or a purely logical one. In this case simply a causality structure is assumed, like in usual algorithmic languages. Algorithmic languages, have been a purely academic area for a long time. The increasing complexity of digital systems and, thus, the need for high level synthesis tools make this level more and more attractive for industry as well. VHDL [VH87] approaching this level makes it even more visible for the industrial practice. Up to now there are only very few commercial systems available for this level. DACAPO III [DA87], partially VERILOG [TH91] and VHDL may serve as examples.

Algorithmic Level:

- Modelling concept: concurrent algorithms
- Timing model: causality
- Observable values: interpreted words

Finally at the *system level* (level 6) details of the algorithmic implementation of system's instruction sets are not considered. At this level the entire system is looked at as a set of cooperating processors. Here the term processor is used in a wider sense to denote a subsystem with an instruction set that enables it to export certain services. A usual processor is the most typical example but channels, device-controllers, etc. fall into the same class. Such a component is characterized by the functions performed by the instructions and the protocol to be used to request a service (an instruction) to be executed. In principle the initiative within such a protocol may be located at the serving device or at the requester. For example, in the case of a usual processor this processor takes the initiative by fetching an instruction (the service it is requested to perform) from memory without explicitly being triggered to do so. In addition, to describe the components of a system and their instruction sets plus protocols, the global interaction of these semiautonomous objects has to be specified. Depending on the kind of system to be described this may be done in a centralized manner or in a decentralized one. In the first case another highly concurrent algorithm serves to specify the global behaviour while in the second alternative the different components,in a totally distributed manner, decide according to certain states or events to request certain services from other components. So the system level can be interpreted as an abstraction of the algorithmic level (centralized alternative) or the register transfer level (distributed way). Both the value domain and the timing model are purely symbolic at this level. There are freely definable types with arbitrary semantics and time is interpreted only to be advanced by causality. The system level up to now is supported by very few commercial simulators. DACAPO III and VHDL are approaching this level while performance analysis tools like HIT [BE86] are addressing the system level as well

System Level:

- Modelling concept: cooperating "processors"
- Timing model: causality
- Observable values: freely definable

3. System Level: General Point of View

Up to now the levels of abstraction have been looked at from the pure electronics point of view. However, pure electronic systems are very rarely built. In most cases an entire system consists of components from various domains like mechanics, hydrodynamics, aerodynamics, thermodynamics, electronics and some software running on the digital part of the electronic components. A digitally controlled car-engine may serve as a typical example where a system consisting of mechanical components, analog electronics, digital electronics running under the control of a sophisticated software have to be combined to perform an operation of high thermodynamical and mechanical complexity.

Consequently at the system level all these aspects have to be considered and the electronics part doesn't usually play the dominant role but a serving one; (it's the engine power a car driver is interested in and not the operation of the electronic controller). The tradition of the various engineering diciplines involved resulted in well defined levels of abstraction for the indivudual disciplines, similar to those discussed above for electronics design.For example in mechanical design there is a well understood abstraction from real mechanical systems up to mathematical models. Looking at the entire system it can be assumed, that during the design process,at the highest level a separation into the different domains of engineering takes place. This partitioning is a highly complex action, up to now almost entirely carried out by human decisions, based on expert knowledge. Once the partition has been decided on and the interfaces have been defined the further design can be carried out within the specific engineering domains. Now the common interfaces may be interpreted individually and modelled in the different areas. This should cause no problem, as from now on a point of view centered at the specific areas is legal. Projected on electronics design after the step of partitioning an electronics system is then obtained, providing an instruction set for software (which is of minor interest for the further steps of electronic design but the central concern for software engineering) and being connected to some peripheral components from a different engineering domain.

There are many degrees of freedom in these partitioning decisions. The design of a digital system may serve as example.*A priori* a solution providing the requested instruction set directly hardwired as hardware is as correct and as obvious as a solution providing the requested instruction set by a piece of software running on a general purpose processor which is available as a piece of hardware. And within the bandwidth spanned by these two extremes a variety of potential, correct and valuable solutions may exist. Only after one of these solutions has been selected the specification of the electronic component to be designed (if not already existing) is obtained.

4. System Level Design

From the above discussion it can be concluded that partitioning seems to be the central design activity at the system level. In order to get something partitioned it is necessary to have a model of the entire system. Modelling therefore is essential, not only but especially at this level of abstraction. This is the reason why this paper concentrates on modelling aspects. Such a model is the initial reaction to a specification of the entire system. This specification should be independent of the solution selected to find an implementation. The model obtained being the first formal document describing the system to be built, some kind of verification is essential. It might be possible to proove analytically that certain aspects of the specification are met. The main activity, however, to check the correctness at the initial model with respect to specification is simulation i.e. experimenting whether the designer's intention is followed. System level simulation therefore is an important design activity. Besides simulation numerous additional analysis activities should be supported, like performance analysis, testability analysis, manufacturability analysis etc. Only when an appropiate set of these design tasks has been performed, the above mentioned activity of partitioning can take place. It produces a couple of individual specifications for the different engineering domains. At the same time the environment for concurrent engineering has to be filled in so that during the entire design process all relevant aspects and parameters can be observed and influenced. So the following design activities are the most important ones at the system level:

a) specification support,

b) system level modeling,

c) system level simulation,

d) system level analysis,

e) system level partitioning,

f) integration into a concurrent engineering environment.

These activities are discussed within this paper in a somewhat more detailed way. The emphasis will be on modelling aspects, however.

5. System Level Specification and Modelling

5.1. General Principles

There is no "one and only" system level modeling method and by the same reason no "one and only" specification method. Requirement engineering being a complex area in itself we shall concentrate on the modeling aspect here. Obviously it depends heavily on how heterogeneous the possible design space is, whether an initial system model can easily be derived from a specification. In relatively simple cases the basic characteristics of a system to be built are fixed, only some parameters have to be supplied. Examples of such approaches are systems like DEBYS [BK91] for logic design. Such systems allow these parameters to be filled in an interactive way where the user is supported by a knowledge based approach. So a search process through a limited (but very large) design space more or less takes place. The result of such a guided search can then be an initial model. Other examples of such an approach are generators for simple DSPs [HI85] or models of RISC processors [NA89, PS90]. This approach however seems to be promising only as long as the design space is homogeneous and limited. It is not surprising that all the examples mentioned are within a single design domain (electronic design in this case). In the general case this "ideal" way of requirement engineering does not seem to be possible. It will not even be possible to formulate homogeneous system models.Contrary to this a multiparadigmatic approach seems to be much more promising. In such an approach for various aspects of a model, different paradigms are used, for each aspect the paradigm that seems to be the most natural one (or that which is most familiar to the designer). In this section therefore a couple of modelling approaches shall just be discussed. It should be noted that a partition of a model into parts using different paradigms doesn't imply the same partition into system components.

5.2. SDL

SDL (Specification and Description Language) [CC88] is a graphical language (with textual counterpart) for specification and description of systems. It is standardized by CCITT where the standard defines two semantically equivalent representations of the language: SDL-GR as graphical representation and SDL-PR as the corresponding purely textual counterpart. It is especially suited for telecom applications. However it is general enough to be used in other areas as well. For example [GV91, LGR92] discusses the use of SDL in electronic design, relating SDL to VHDL.

SDL has the communicating sequential processes point of view i.e. a system is described in SDL as a set of concurrent processes that communicate via shared channels by (asynchronously) sending and receiving messages. The asynchronous nature of SDL communications implies that buffers have to be assumed at each communication channel.

SDL supports different views of a system description:

- a structural view, supported by
  - **Block Interaction Diagrams (BD)**
- a communications view, supported by
  - **Sequence Charts (SC)**, and
- a concurrent behavioural view, supported by
  - **Process Diagrams (PD).**

The *BD*s are just usual hierarchical schematics as used at each level of electronics CAE. The edges in a BD represent the communication channels. Attached to such a channel description is a list of messages that can be sent over this channel. By identifier equality a *BD* is connected to the dynamics (the behaviour) expressed by attached *PD*s and *SC*s. *PD*s and *SC*s both describe the system's behaviour. In a *SC* the global view is stressed. It is described which communication sequences can be observed from the outside world if it is abstracted from the processes internal to the communicating objects. The inverted point of view is described by a *PD*. It abstracts from the global communication structure by just describing what messages are sent and received. On the other hand the process where these atomic communication actions now play a role is specified precisely.

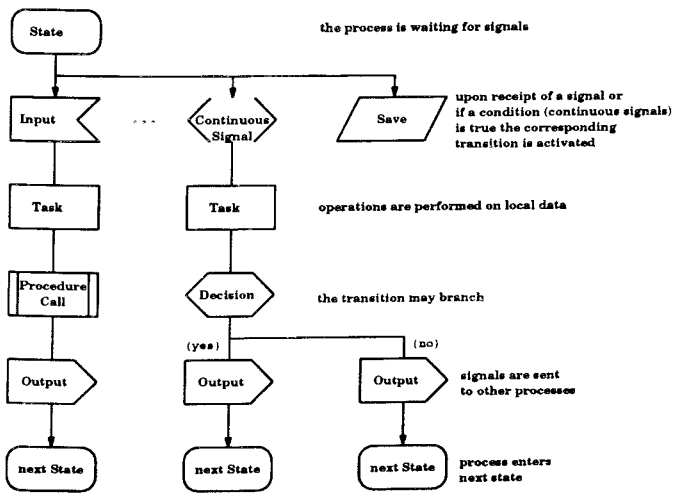

Fig. 5.2.1: SDL PD

The method used is an extended finite state machine. Assume that a process is in a certain state *S*. It waits there until it finds an acceptable message in its input queue. For each state there may be a specific set of acceptable messages. If a message arrives at a process which is not in a state to accept it, the message may be discarded or saved, depending what has been specified for this situation. When an acceptable message has been found in the input queue, a state transition takes place. During this transition local data may be modified and messages may be sent to other processes. The next state reached by a transition may depend on actual values of variables. Autonomous state transitions are possible as well, based on *continuous signals*. In SDL there is the concept of single ownership of data by processes (local data). Owned data may be explicitly made visible to other processes.

The multiview approach of SDL makes individual descriptions very easy to understand. To get a complete understanding of an entire system, however, it is necessary to combine several subdocuments and to understand the intercorrelation. Like Statecharts (see 5.6) SDL supports only asynchronous communication but synchronous communication (rendezvous) can easily be simulated by this concept by explicitly describing a handshake procedure. SDL is in practical use worldwide. Recently extensions towards object orientation (*SDL'92*) have been proposed [SDL92].

5.3. SADT and SA

SADT (Structured Analysis and Design Technique) consists of two parts:

- a graphical language SA (Structured Analysis)
- a methodology DT (Design Technique)

While in SA the structure and communication of objects (which are specified in any other language), is described, DT serves as an instruction on how to use SA. SA looks at a system from the point of view that there are "things and happenings" [ROS77]. Such things, happenings, and their interaction are described in SA by diagrams. Such a diagram is a finite, directed, edge- and vertex-annotated graph. There are two basic interpretations of such graphs: if the nodes (drawn as rectangles) are associated with activities and the arrows with data such a graph is called *activity diagram*.

One identifies a horizontal data stream and a vertical control stream, where *control* denotes such data that directs the activity to perform specific operations while *mechanism* denotes specific techniques to be applied. The second interpretation of the SA graphs is an inverted view. In this case, the rectangles are used to denote data and the arrows the activities envolved. Such graphs are called *data diagrams*. Typically a data object is drawn as indicated in Fig. 5.3.1.

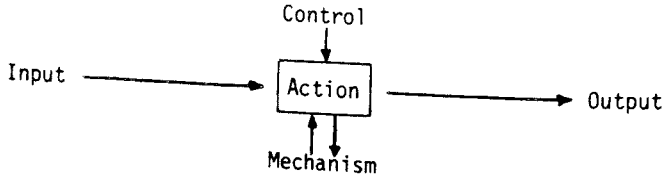

Fig. 5.3.1: Basic element of SADT data diagram

In this case, the producing and consuming activities of a data object are drawn in the horizontal direction while the controlling activity and the storage means are shown vertically.

Both diagrams may be decomposed hierarchically. There are no sequencing contructs in SA. Activity diagrams therefore are strict dataflow diagrams, that do not contain information about a control flow. A control flow that is consistent with an activity diagram may be constructed using specific sequentialization techniques. DT provides methods to support structured understanding of complex systems; teamwork; communication of intermediate results; control of adequacy, completeness and quality; management. To achieve this, DT proposes certain fixed processes for system design.

SADT is relatively vague in its description. It may be applicable in early design steps to structure thinking. However, it is very hard to obtain precise and executable specifications of systems to be designed. So it is not surprising, that SADT tools are restricted to those which support documentation.

Structured Analysis (SA) [De78] is another dataflow-oriented method for requirement engineering. Here SA denotes the entire method, not to be confused with the SA-language of SADT. In the present context however, an extension of SA, called Real Time Structured Analysis (RT/SA) [WM85] will be discussed. This extension includes control-flow specifications. A system description in SA consists of a set of *dataflow diagrams*, a *data dictionary* and a set of *transformation descriptions*.

A dataflow diagram is a graphical representation via a directed graph of functions with their interfaces and dataflows. There are three types of nodes in such a diagram, circles to denote actions/processes, twin horizontal bars to denote data stores (files) and rectangles to represent terminal nodes (springs and sinks). The flow of data is represented by arcs. Fig. 5.3.2 shows the different symbols.

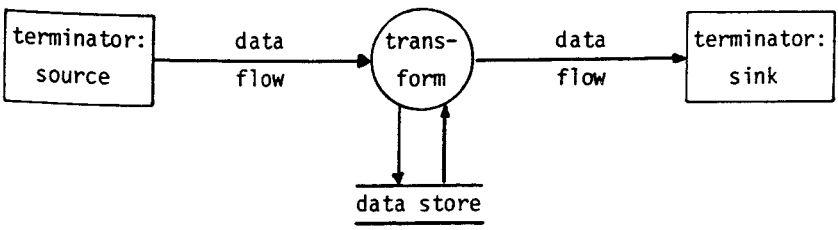

Fig. 5.3.2: Graphical symbols of SA

Edges and vertices of SA dataflow diagrams may be annotated with identifiers which can be referred to in additional SA documents. The first type of such an additional document is given by the *data dictionary*. In this document the terminology of the project is described and further details of the objects denoted by identifiers in the dataflow diagrams are provided. Documents describing SA propose to use regular expressions and plain text for this purpose. The second type of additional documents is given by *transformation descriptions*, often called *minispecs*. A minispec defines in further detail the internal operation of a function within a dataflow diagram. No global information can be provided in a minispec. Minispecs may be formulated in pseudocode, decision tables, plain text or any other formalism. For

example in [TLK90] it is proposed to use VHDL to formulate SA minispecs.

RT/SA adds to the above concepts *controlflow diagram* and *control specifications*. The two views (dataflow and control flow) are bound together by using nodes with the same identification. This may result in isolated nodes in one of the diagrams as, e.g. there may exist activities in the controlflow diagram that are not embedded in the dataflow. Such nodes then appear as isolated nodes in the dataflow diagram. A control flow diagram has exactly the same syntactical structure as a dataflow diagram, only its semantics is different. In the same way as there are minispecs, a control flow diagram may have control specifications attached. Again, a specific control specification is a purely local one, it specifies, how one node of a control flow diagram has to be interpreted. Various techniques are possible for control specifications, including VHDL [TLK90].

Like SADT, SA supports hierarchical descriptions by stepwise refining nodes of dataflow/controlflow diagrams to such diagrams. There are some tools available for SA (RT/SA) including checking editors and analysis tools.

5.4. GRAPES

GRAPES-86 is a graphical language to support all aspects of system design. It has been designed by Siemens Nixdorf Informationssysteme AG [HE90] to be used for projects of high complexity. Like SDL, GRAPES makes use of a couple of correlated diagrams to describe an entire system. In GRAPES this multi-representation approach is even expanded by introducing additional types of diagrams. GRAPES is intended to combine principles of IORL (Input/Output Requirements Language) [TB84], SA (Structured Analysis) [DE78], SADT (Structured Analysis and Design Technique) [YC79] and SDL and approaches an object oriented view by looking at a system as communicating and cooperating objects.

GRAPES has the "Cooperating Processors" point of view which in section 2 has been identified to be the basic modelling principle of the system level. So a system in GRAPES is modelled by a structured set of individual parts which interact to achieve the functionality to be provided by the entire system. These parts are called objects in GRAPES. They are semiautonomous and interact only via communication channels. Objects may be decomposed into structures of subobjects which in turn are objects. So arbitrary hierarchies can be built. GRAPES has an elaborate graphical notation, where concepts are inherited from well established notations as far as possible.

Communication Diagrams (CD) serve to describe the static structure of objects. Thus, GRAPES CDs are comparable to SDL BDs. It is described how an object is decomposed into subobjects and what communication relationships are established via communication lines.

For each communication line an *Interface Table (IT)* has to be provided. This diagram specifies the channels to be used, the data types to be transmitted and the kind of communication (synchronous or asynchronous). The dynamic behaviour is specified using *Process Diagrams (PD)*. These process diagrams are inherited from SDL and have nearly the same syntax and semantics as in SDL. Slight modification originate e.g. from the existence of synchronous communication in addition to asynchronous.

The local data of processes are specified via *Data Tables (DT)* and the local call interfaces of procedures and functions are described using *Specification Diagrams (SD)*.

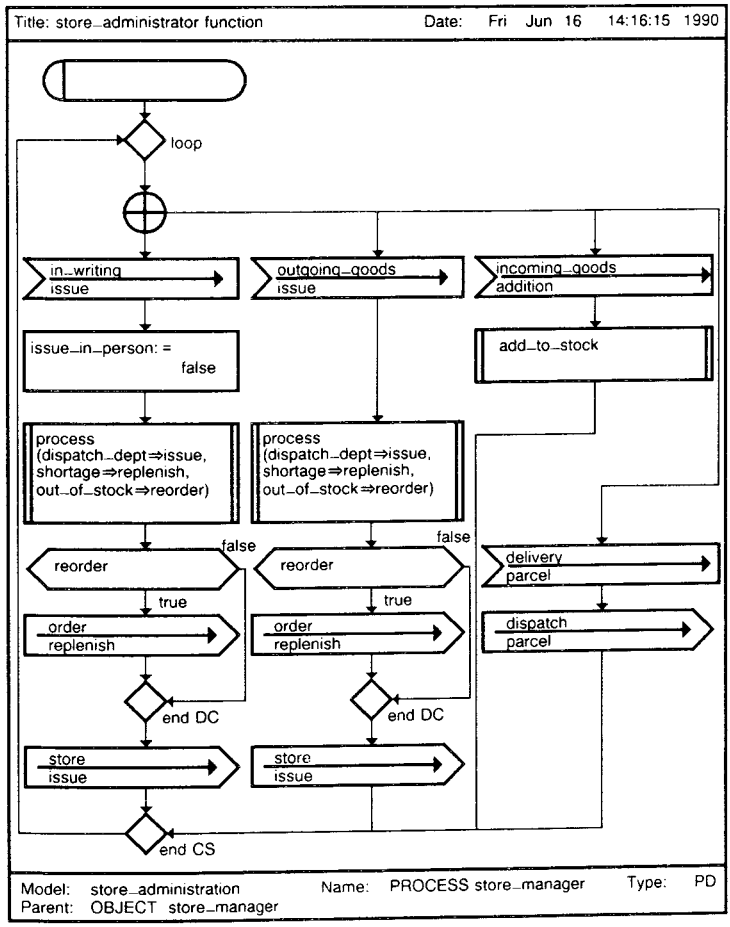

Fig. 5.4.1: GRAPES PD

In addition to these specifications there exist *Data Diagrams (DD)* to model complex data structures. DDs are a version of E/R diagrams using an easy to understand graphical notation.

Finally *Hierarchy Diagrams (HD)* are used to represent the interrelationships between all the documents used to model a system.

So GRAPES is an attempt to combine SA-like data flow diagrams, SDL-like process diagrams and the entity relationship data modelling concept to one single well defined (concerning syntax and semantics) modelling concept. Its dynamic modelling concept (communicating sequential processes) is inherited from software engineering and covers a wide spectrum of applications. For extremely parallel systems, this approach may tend to become a little bit unwieldy.

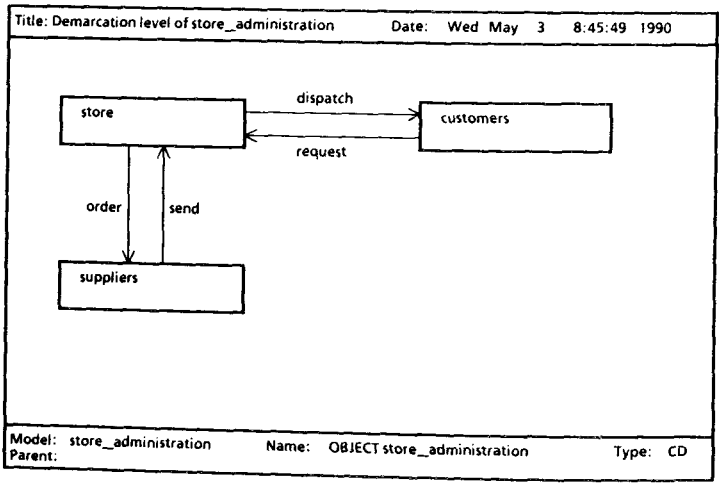

Fig. 5.4.2: GRAPES CD

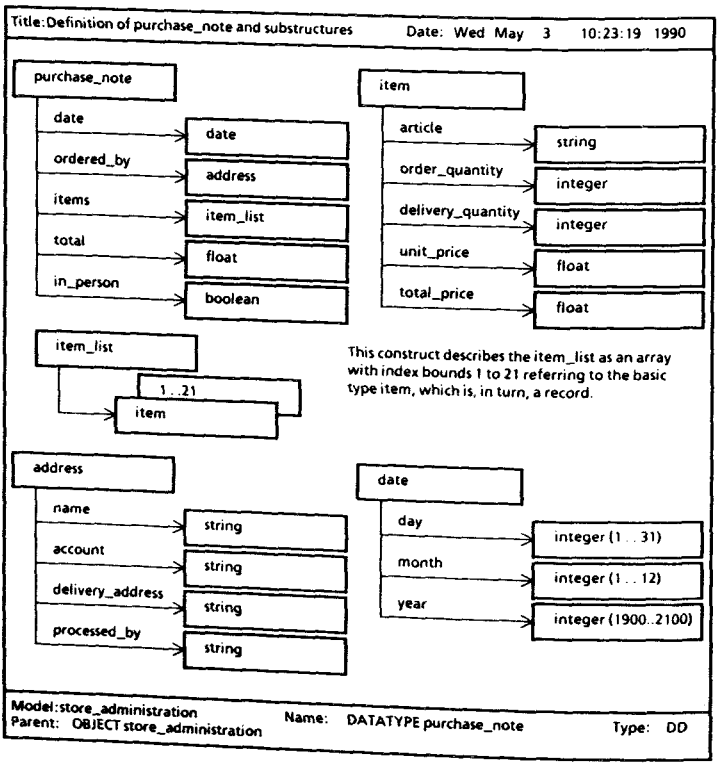

Fig. 5.4.3: GRAPES DD

5.5. Logic Programming

Specifying a system by means of rules to be respected seems to be a natural approach. This method is not very powerful alone as long as there is no inference mechanism that allows one to decide what rules are applicable and by which further knowledge can be deduced. A very general inference mechanism is given by *unification.* Therefore PROLOG [CM81] and its derivates become a useful language for specific aspects of system level specification and modelling. PROLOG is a language which is well suited to describe a calculation by means of the intended result, independently of intermediate steps. If processes where these intermediate steps are of interest rather than a final result (which may not exist) the use of PROLOG makes no sense as by the nondeterministic *backtracking* mechanism illegal intermediate states may be reached that cannot be communicated to the outside. For this purpose *stream parallel committed choice* languages like PARLOG [CG84], GHC [UE85], or FCP [SH87, GK90] are well suited. FCP may serve as a typical example. A FCP clause looks like $H : -G_1, G_2, \ldots, G_n \mid B_1, B_2, \ldots, B_k$. H is the *clause head.* A goal can be unified with this head as in PROLOG. The G_i are *guard predicates.* The conjunction of these G_i has to be true for the *clause body* to be executed. If so the |-symbol (*commit*-symbol) is passed irreversably i.e. from now on there is no back-tracking. The *body predicates* B_j can be interpreted as processes that are initiated concurrently by the commit operation. Fine grain communication between the concurrent processes is carried out by shared variables that might be specified as *read only variables* in certain processes. This means that such a process is not allowed to instantiate such a variable but has to wait until this has been carried out by a process that has the right to do it. Concurrent logic programming has been used for hardware specification [SU85, WS87]. Efficient parallel implementations are available today [GH91,GL91]. FCP and their role in system level specifications will be discussed in more detail in section 6.

5.6. Statecharts

Statecharts [HA 87] start from usual state diagrams of FSMs. In order to handle complexity, first of all a concept of *hierarchy* is added. This is done by allowing a state to be decomposed into an FSM as well, and so on. Graphically, the embedded FSM is drawn just inside the *macrostate.* This concept makes it necessary to introduce means for specifying in which internal state such a macrostate has to be started when activated. This may always be the same one (default initial state) or be dependent on the internal state a macrostate has been when this macrostate has been deactivated. The first case is denoted by a single state transition arc that originates from an isolated location inside the macrostate and ends at the initial state. The latter situation is supported by the so called *history mechanism* that may be extended recursively to deeper hierarchy levels. The history mechanism is denoted by introducing a state symbol labelled with H inside a macrostate (or H^* if the history mechanism has to be continued recursively until the lowest embedded level. This state symbol is the destination of a state transition arc originating from a state outside the macrostate i.e. the one from which the macrostate is activated. The second concept added by Statecharts to FSMs in order to handle complexity is *concurrency.* Several Statecharts are now allowed to operate concurrently i.e. when the macrostate they are embedded in is activated, more than one internal FSM is activated. Graphically concurrently active FSMs are denoted by dividing the embedding macrotransition using dashed lines and including a Statechart in each of the partitions. Introducing concurrency always makes it necessary to introduce synchronization and communication concepts. In the case of Statecharts this is done

by a *broadcasting* mechanism, i.e. by asynchronous communication. Broadcasting is described by the usual edge annotations of FSMs, i.e. an annotation of the form l_{in}/l_{out} denotes, that this transition takes place if event l_{in} happens and causes event l_{out} to happen. These events are distributed all over the concurrently active FSMs.

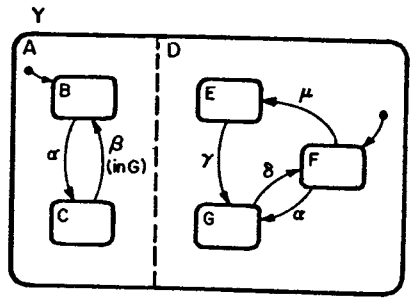

Fig. 5.6.1: Example of a Statechart

5.7. Extended Petri Nets

Petri Nets [Re85] are a technique that has been used for the specification and analysis of (concurrent) systems for decades. As a result a relatively rich set of mathematical results is available.

Petri Nets constitute a graphical model of activity flows. The graphical representation is given by *Petri Net graphs*. A Petri Net graph $PG = (P, T, E)$ consists of a finite set P of *"places"*, a finite set T of *"transitions"* and directed edges connecting places with transitions and vice versa. Places represent condition variables and are drawn as circles while transitions model actions. They are drawn as horizontal bars (or rectangles). Activity is introduced into a Petri Net via the concepts of *marking* of places and *firing* of transitions. Each place may contain an arbitrary set of *tokens*. A transition fires if each of its input places is marked with at least one token. If this is so, it fires. This consists of removing one token from each of its input places and adding one to each of its output places.
Various classes of Petri Nets (defined by restrictions on the graph topology) have been investigated and a couple of mathematical results have been obtained that are valuable in system design. Ordinary Petri Nets however, tend to become extremely unwieldy if large systems have to be modelled. Numerous extensions have therefore been discussed in the literature. Some of these extensions concentrate directly on the needs of hardware design, e.g. the approaches related to DACAPO [Ra80] or Cascade [LM92]. As extensions of Petri Nets will be discussed in more detail in section 6, only the concept of *Structured Petri Nets* (S-Net) will be briefly introduced here.

If Statecharts may be characterized by the sequence of adding to FSMs hierarchy first and concurrency afterwards this sequence is inversed in the case of S-nets [CK81]. They start from Petri nets, i.e. the concept of concurrency has already been added to FSMs by the usual extensions introduced by Petri nets. The remaining task to find a consistent concept for hierarchy (a problem that turned out to be a complicated one) has been solved by S-nets in a very elegant way. Here each

transition may be replaced by an entire S-net with a "flat" Petri net being an S-net as well. A *macro-transition* becomes firable by the same condition as a usual one. Internally the firing of a macro-transition means that the (always identical) initial marking is taken. Starting with this marking the local S-net becomes active and remains active as long as it is live. By this internal net becoming dead the macro-transition plays its token game in the same way a usual Petri-net transition does. Each Statechart can be simulated by an S-net and a subclass of S-nets that covers all cases relevant for practical applications can be simulated by statecharts [SU90]. So for practical applications they can be looked at as equivalent. This is interesting as Statecharts introduce hierarchy by decomposing states while S-nets decompose transitions. Which concept is the more natural one depends on the special situation. The history mechanism is missing in S-nets as originally defined by L. Cherkasova, but can easily be added.

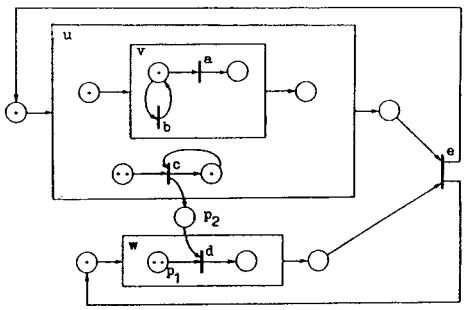

Fig. 5.7.1: Example of a Structured Petri Net

5.8. Programming Language Paradigms for System Level Modelling

System modelling by **functions** is a very natural way with a long tradition. By systems of differential equations very complex systems can be described in a concise way. The v.Neumann-paradigm unfortunately replaces equation by **assignment**. As for the numerical calculation of equation systems v.Neumann computers have to be programmed, this assignment point of view entered areas where it is completely inadequate. *Functional modelling* is very natural in any kind of continuous systems. In the area of electronics, analog devices are a typical example. But also in all kinds of engineering **continuous systems** are investigated and mostly described by means of **differential equations**. So system level modelling without supporting a functional point of view rarely makes sense. An excellent example for an approach to cover wide areas of electrical systems by functional techniques is GLASS [SE90]. Functional programming is not only a very natural approach. It also opens the way to the long tradition of mathematics. By analytical and algebraic methods formal proofs and transformations can be carried out fairly simply on functional specifications while the same is very complicated in the imperative domain. Therefore it is not surprising that functional programming is playing more and more an important role in software engineering as well. This trend is increased by the observation that the parallelisation of functional models is much easier compared with imperative ones. A language that influenced functional languages in the software domain is ML [MT90] while LISP can be looked at as the classical functional language.

If there is a paradigm that has the potential to cover most areas of system engineering, *object-orientation* may be the best candidate. By the principle of describing systems by a set of *objects* that are incarnations of *object types*, i.e. *classes*, the structural aspect is covered very well by the OO approach. This is achieved even better if the principle of *inheritance* present in most OO languages is used. By this principle hierarchies of classes can be built. Fairly complicated systems of classes can be described in a concise way by this approach. Objects are not just passive entities. They offer *methods* that can be requested by other objects to be performed. Together with the principle of *polymorphism* this is a very powerful concept of abstraction. Polymorphism in this context means that in order to request a method just a message is sent to the offering object. By the nature of the message the offering object can decide how the request can be satisfied. Object orientation has a nice mathematical background by the theory of *abstract data types* (ADT) [GH78]. An ADT *D(S,E)* is given by a *signature S* and a set of *equations E*. A signature *S(sorts, ops)* is given by a set of *sorts* (domain identifiers) and a set *ops* of operations defined on these sorts. The signature specifies the syntax of the ADT. The semantics is given by the set of equations to be respected. A couple of OO languages is available today with C++ [ST86] playing a dominant role just by being inherited from C. Recently efforts have been made to make concurrent OO languages more practicable [AH90]. This of course is essential if all aspects of system engineering have to be covered, as most aspects are concurrent by nature. Another area to be attacked is the specification of the *protocols* to be used between requesting objects and serving ones [MR89]. Traditionally in OO languages nothing is specified for this purpose. It is just assumed that any message sent from a requester is obviously understood by the server. Technical systems behave in a completely different way.

5.9. HDL Paradigms for System Level Modelling

CHDLs like VHDL have been designed to cover a specific aspect of system (though efforts are underway to define VHDL extensions to support analog design as well) engineering, i.e. the design of electronic hardware. In the case of VHDL this area is further restricted to digital systems and a bandwidth of abstraction levels that reaches from partly switch level to partly algorithmic level. As dedicated languages, however, CHDLs should be the most adequate means to describe objects within their domain. As we have already identified that a multiparadigmatic approach seems to be the most promising one for system modelling, CHDLs have their natural role in this context. The approach of SpecCharts [VN91] is an excellent example for this idea. Here the entire system is specified by a Statechart-like formalism, the fine grain functionality of the states, i.e. the operations to be performed in the states, are specified in VHDL. The approach of [TLK80] to use VHDL as language to formulate minispecs within RT/SA is similar. Another approach to add system design capabilities to VHDL is VAL [AG88]. Here additional information is added to allow reasoning about descriptions provided. There are other CHDLs that offer more direct support for the system level. DACAPO III [DA87] is an excellent example for this. In contrast to VHDL this language supports

- functional programming,
- implemented abstract data types,
- module concept,
- algorithmic concurrency

in addition to the scope covered by VHDL. Therefore it is not surprising that a couple of proposals to enhance VHDL towards system level descriptions have been made by researchers that are familiar with DACAPO III [Glu90, OC90]. DACAPO III was also the basis for ODICE [MR89] an approach to introduce real object orientation, including protocol specification, into a CHDL. A perfect idea with respect to the intended polyparadigmatic approach is the concept of CONLAN [PB83] to provide a deduction system for building modelling languages instead of just defining one language. A revival of CONLAN might be a very promising way of approaching a framework for system level modelling.

6. Towards a Unified System Level Modelling Technique

6.1. General Principles

When looking for a specification method that covers both control and calculation, that is able to deal with concurrency and that has a rigid formal foundation, Predicate/Transition nets (Pr/T-Nets) [GL81] seem to be a well suited candidate. In contrast to Statecharts [HA87] they combine the representation of control and data-operations in one single diagram and in contrast to the Z-notation [SP92] they offer a natural support to express concurrency. Due to the easy to follow graphical representation Pr/T-Net specifications are relatively easy to understand. On the other hand they have the computational power of Tuning Machines [GL81].

Unfortunately in the original definition of Pr/T-Nets no mechanism to handle hierarchy has been provided. However it has been investigated [Ku92, Ki92] how the hierarchy concept of Structured nets [CK81] can be applied to Pr/T-Nets. But even with hierarchy added, another basic specification technique is missing: recursion. In this paper it is shown that recursion can be added in a relatively easy manner when using "colored" tokens [JE91]. With these extensions an impressive specification language with high expressive power has been obtained.

There is an increasing demand for executable specification methods and (related to this) for support of rapid prototyping. Of course it is possible to build a dedicated simulator for extended Pr/T-Nets. Such a project, however, is cumbersome as a couple of problems, that prove to be hard to solve efficiently, are present, especially concurrency, recursion, finding interpretations that allow transitions to fire. When looking more closely at these problems it can be seen that most of them are present when implementing Flat Concurrent Prolog (FCP) [SH87]. FCP is the most powerful representative of the class of committed choice stream parallel logic programming languages. Other members of this class include Flat Parlog [FT87], Flat Guarded Horn Clauses (FGHC) [IMT87] and KI/1 [Na92]. Common to all these languages is that they constitute a subset of Concurrent Prolog [SH86] that can be implemented efficiently. In the case of KL/1 such implementations exist on Unix-Workstations and on dedicated hardware, ICOT's Parallel Inference Machine (PIM), while efficient implementations of FCP on Unix-Workstations and on large Transputer networks have been reported recently [GL92]. So it seems to be an interesting question whether FCP can be used directly to execute extended Pr/T-Nets. On the other hand, FCP programs look somewhat unfamiliar to most programmers and are thus hard to understand for them. The question therefore arises, whether extendend Pr/T-Nets can be used as graphical representation of FCP. The first question is investigated in this paper and a positive answer is obtained.

6.2. Predicate/Transition Nets

Mathematically Pr/T-Nets form somewhat complicated objects. Therefore only a short informal introduction shall be given here.
The first basic difference between ordinary Petri Nets and Pr/T-Nets is that in Pr/T-Nets tokens are individuals while in ordinary nets only their count on places is of interest.
In Pr/T-Nets a token is an instantiated object of a certain type. Assuming an underlying Petri Net graph $PG = (P, T, F, B)$, P finite set of "places", T finite set of "transitions", $P \cap T = \emptyset$,
$F \subset \{(t,p) \mid t \varepsilon T, p \varepsilon P\}$,
$B \subset \{(p,t) \mid p \varepsilon P, t \varepsilon T\}$,
a global making M now is a bijective mapping from the set of places P onto a partition of the set of instantiated tokens,
$M : P \rightarrow \{To_i \varepsilon \mathbf{P}\ (\mathrm{T}_o) \mid T_o =$ set of instantiated tokens, $T_{o_i} \cap T_{o_j} = \emptyset$ *if* $i \neq j, \cup T_{o_i} = T_o\}$.

i.e. tokens are instatiated only as part of the marking of a place and each instantiated token marks only one single place.
Input arcs of a transition $t \varepsilon T$ are labelled with typed variables. A transition is firable only if there is a valid interpretation of the set of these typed variables using currently instantiated tokens in the respective places. Equally named variables attached to input arcs of one transition have to be substituted by the same values for an interpretation to be valid.
Transitions have attached a predicate and a token mapping. A transition t fires under a specific interpretation only if its predicate is true for this interpretation. So looking for a valid interpretation may be interpreted as looking for sufficient syntactically correct data while testing the predicate means testing whether semantical restrictions are met.

If a transition fires, it destroys the input tokens that constitute the interpretation it is reacting on and calculates (i.e. instantiates) tokens on its output places. By labelling output arcs with typed variables, values can be routed individually to token instantiations.

Example 1:

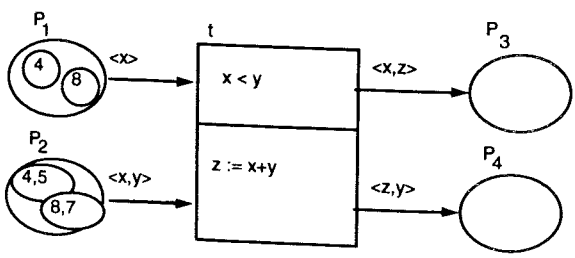

Fig. 6.1: Pr/T-Net before firing

Assume the type of all variables to be integer. There are two valid interpretations: $(x \rightarrow 4, y \rightarrow 5)$ and $(x \rightarrow 8, y \rightarrow 7)$ but only the first one is accepted by t for firing, as the transition's predicate requires $x < y$. As result of the firing the following marking is obtained:

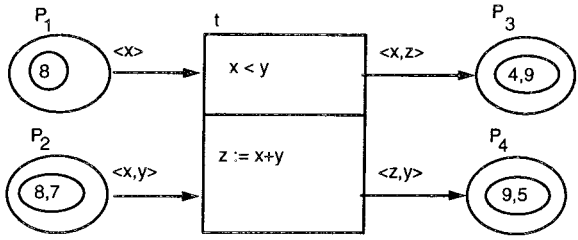

Fig. 6.2: Pr/T-Net after firing

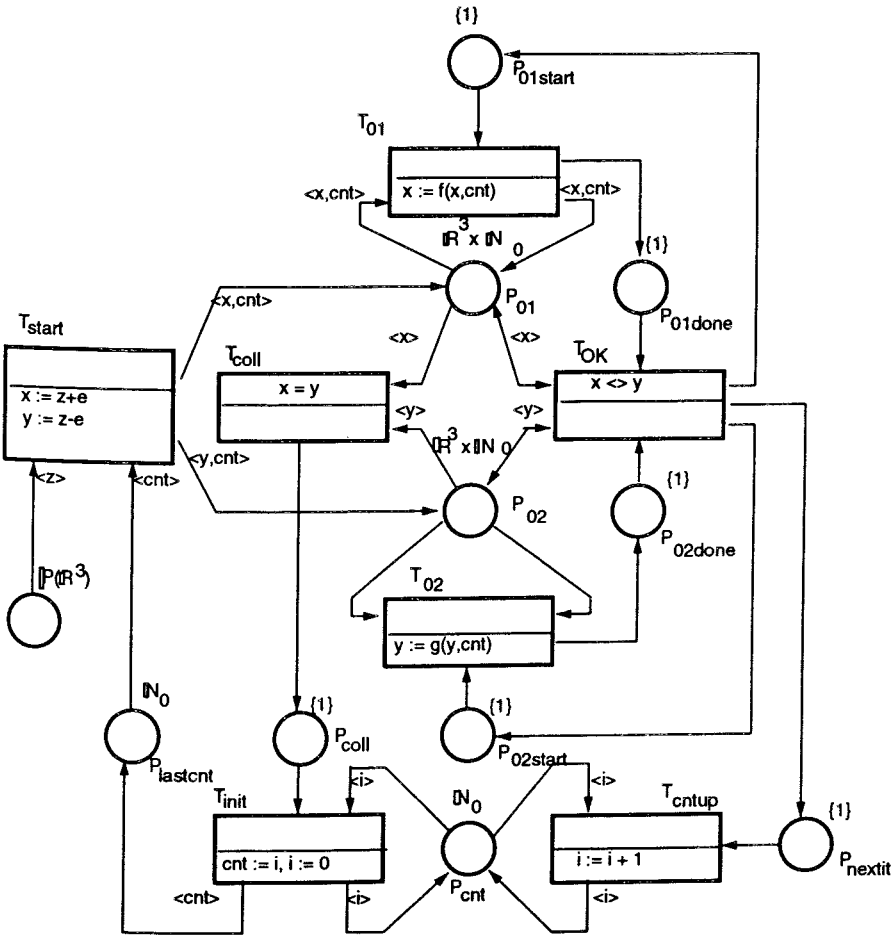

Fig. 6.3: The Collision Problem as Pr/T-Net

Only basic concepts have been described here. Various modifications/extensions are possible. For example in [KU92] set operations have been included in the interpretation/firing mechanism or in [BOP89] linear factors and sums of variables are included in the labelling of arcs. Such extensions make modelling more convenient. This section is closed by a more realistic application example: Assume a system has to be modelled, where two objects are flying in a three-dimensional space, starting from two points that are closely neighboured but not identical. The two objects decide independently where to go by iteratively calculating the next position as a function of the present position and an additional integer argument. The iterations however are kept synchronized between the two objects. The number of iterations is counted. If the two objects happen to collide, the system is reset. This means that the iteration count is handed over to be used as the additional integer argument *cnt* in the next try, and then is reset to zero. Starting close to an externally given position the two objects try again.

Explanation:

The calculation of the next position of object O1 (O2) is carried out by transition $T_{O2}(T_{O2})$. Both of these transitions can fire only if their arguments are present (P_{O1}, P_{O2}) and the next iteration may start $(P_{O1start}, P_{O2start})$. Transition T_{ok} fires after T_{O1} and T_{O2} have finished their calculations (reported via P_{O1done} and P_{O2done}) and the two new positions are not equal. By its firing, T_{ok} not only re-enables T_{O1} and T_{O2} (via $P_{O1start}$ and $P_{O2start}$) but also causes (via P_{nextit}) the transition T_{cntup} to count up the iteration counter. (Notice that T_{ok} is reading P_{O1} and P_{O2} in a non-destructive manner as it returns the unchanged values). As soon as the two positions become equal (asynchronous event!), T_{ok} is no longer enabled but T_{coll} will fire immediately. It removes the tokens from P_{O1} and P_{O2} and initiates a re-initialization via P_{coll}. Re-initialization is performed by T_{init} by reporting the current value of the iteration counter *cnt* to $P_{lastcnt}$ and resetting the iteration counter to 0. Aside from possibly different values of Z and *cnt* the initial marking has been restored now, and a new try is possible.

6.3. Coloured Pr/T-Nets

Colouring tokens does not increase the modelling power of Pr/T-nets as this concept is implicitly included in the model. So it is for convenience only, that coloured tokens are introduced here. The notation of coloured tokens will make it easier to introduce recursion later.

In a coloured Pr/T-Net there exists a (finite or infinite) number of classes (colours). Each instantiated token belongs to exactly one class. (For convenience one may introduce a joker-like superclass each instantiated token belongs to, to which by introducing more sophisticated class hierarchies even more elaborate colouring mechanisms may be introduced. In this paper the simple case of single class membership will be assumed.)

A transition t is now activatable only if there exists a colour such that t would be activatable if only tokens of this colour existed. Note that now a transition t may be activatable concurrently and independently for different colours. So it makes sense to fire a transition independently for different colours. By this the concept of a "coloured firing" is obtained where a transition t which is activated with respect to a specific colour c of tokens, fires exactly if only tokens of this colour existed. It may, however, change the colour of the tokens during the token game, i.e. the colour of the input tokens may be mapped on a different one of the output tokens.

Within the framework of Pr/T-Nets colouring is introduced simply by replacing each object-type by a pair (type, colour). Then the basic search for a valid interpretation of Pr/T-Nets results in the colour-oriented activation rule, initiating a coloured firing. The colour-mapping can easily be included in the mapping part of the transition.

Example 2:

For the little situation of Fig. 6.4 it may be assumed, that there are the colours red, green, blue to be considered. So in P_1, P_3 we now have tokens of type $(\mathbf{N} \times colour)$ with $colour = \{red, green, blue\}$ while in P_2 and P_4 we have tokens of type $((\mathbf{N} \times \mathbf{N}) \times colour)$. The edge annotations would have to be augmented by a colour-component of the variables as well, but as we know we are in the context of coloured nets, this can be ommitted. The colour-mapping, however, has to be included into the transition's mapping part. It may be assumed that the function cf: $colour \rightarrow colour$ is defined by $red \rightarrow green$, $green \rightarrow blue$, $blue \rightarrow red$.

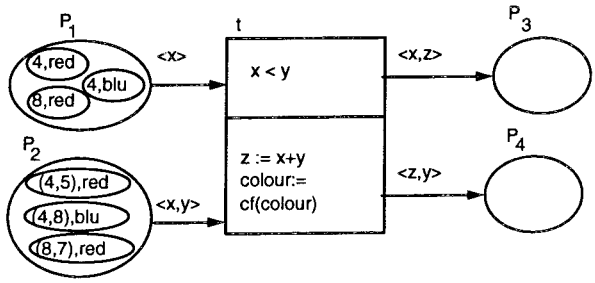

Fig. 6.4: Coloured Pr/T-Net before firing

For both colours, red and blue, valid interpretations exist;
for *red*: $(x \rightarrow 4, y \rightarrow 8)$ and $(x \rightarrow 8, y \rightarrow 7)$,
only the first one being in accordance with the predicate
for *blue*: $(x \rightarrow 4, y \rightarrow 8)$, being in accordance with the predicate.
So two independent coloured firings may occur,
resulting finally in the following situation:

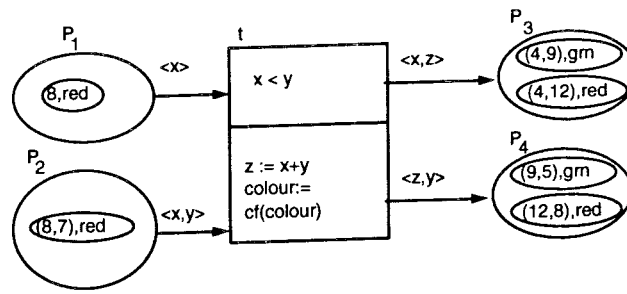

Fig. 6.5: Coloured Pr/T-Net after firing

Colouring may also be interpreted as a specific folding technique where multiply used transitions calculate different streams of data independently.

6.4. Hierarchy

Besides decomposition (which is related to concurrency) hierarchy and recursion are the basic techniques to conquer complexity. Decomposition and concurrency are a basic principle of Pr/T-Nets. But hierarchy and recursion have still been missing up to now. They shall be introduced in this section and the following one.

The way we are introducing hierarchy into Pr/T-Nets is heavily influenced by the approach of Cherkasova/Kotov [CK81] for hierarchical ordinary Petri Nets. (see 5.7) These concepts shall now be described in more detail. To describe the structural aspects, the concepts of [Ki92] are used.

The underlying structural concept of Pr/T-Nets is the Petri Net graph $PG = (P, T, F, B)$. To avoid complicated graph grammar approaches, a simple port/port-map technique similiar to VHDL [VH87] is used here. Each transition's or place's input/output edge may serve as a port, where each port may be of one of three types $\{in, out, inout\}$. The set of all ports of a net forms its interface. Graphically a place-port is drawn as a solid dot, a transition-port as a solid rectangle and the port-type is denoted by an arc between the port-symbol and the node being a port. As in Pr/T-nets the individual edges of a transition t have individual meanings, they are identified by $t.a$, where a is the edge's attribute.

Example 3:

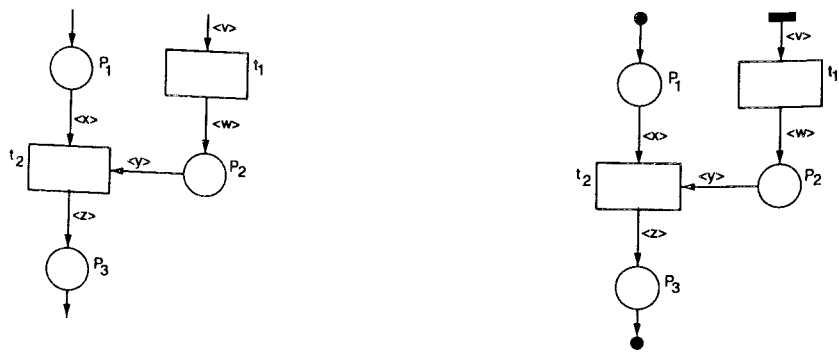

Fig. 6.6: Petri Net Graph and Petri Net graph with interface

Hierarchy is now introduced by allowing a transition of a (hierarchical) Petri Net graph to instantiate a (hierarchical) Petri Net graph with interface. By this process, a copy of the graph to be instantiated is made and located within the instantiating transition. After this copying process the ports have to be connected. This is done by a port-map statement that connects ports with local nodes (places or transitions) or with ports of other instantiated nets. Of course only such port-map statements are valid that result in a valid Petri Net graph, i.e. only places may be connected to transitions and vice versa.

Example 4:

Assume the nets given by Fig. 6.7 and Fig. 6.8 (i.e. places P_1, and P_4 are marked with one token each. In this example we assume tokens of no individual meaning, i.e. normal Petri Nets. Therefore the transition edges are not attributed in this example).

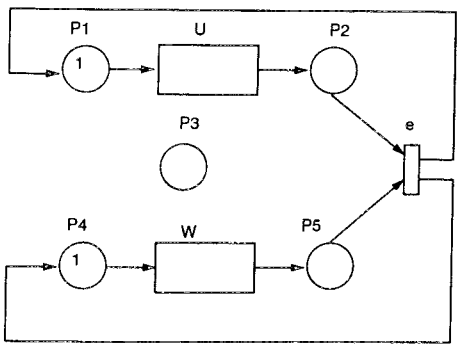

Fig. 6.7: Petri Net graph without isolated transitions

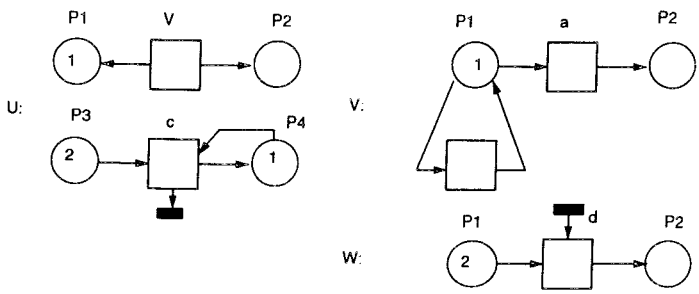

Fig. 6.8: Definition of subnets

If transition u instantiates net U, transition w instantiates net W, transition $u : v$ instantiates net V, and there is the port-map $(u : c, w : d)$ then the hierarchical Petri Net as shown in Fig. 6.9 is obtained.

Up to now a concept has been followed up, that is comparable to allowing global variables in programming languages. If "global variables" are not to be allowed, nets imbedded in macro-transitions can communicate with the outside only via ports of this macro transition. For this purpose named and typed ports at macro-transitions are introduced. Such a port may be of transition-kind if it is to be identified with a transition-port of the interface of the embedded net or of place-kind in the other case. In all cases it may be *in*, *out*, or *inout*.

Graphically a port of a macro transition is drawn as a solid rectangle (transition-kind) or solid circle (place kind) on the border of the macro-transition's symbol.

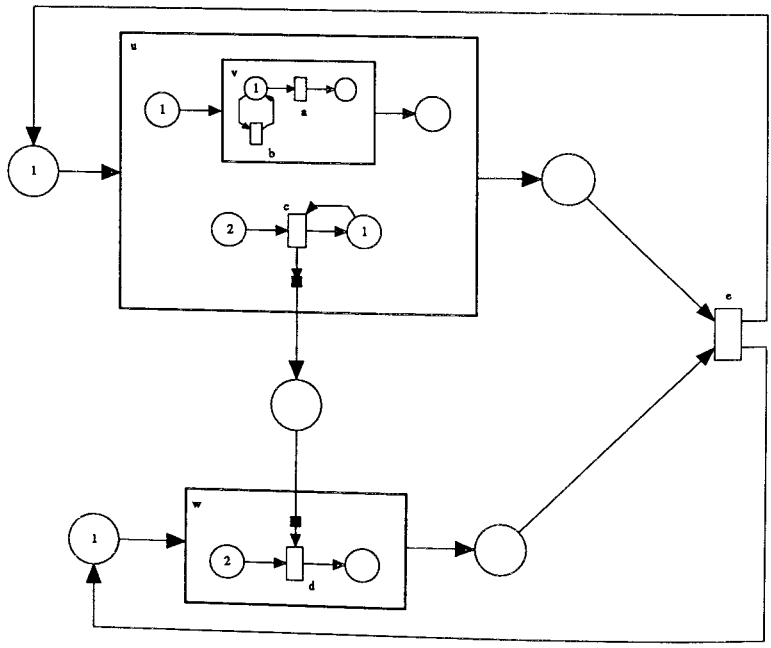

Fig. 6.9: Regular Petri Net graph

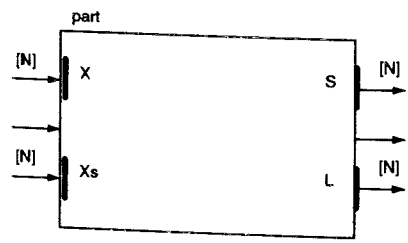

Fig. 6.10: Macro-transition with formal ports

Example 5:

In Fig. 6.10 the macro-transition *part* has four ports, all of kind transition port, i.e. internally they have to be identified with transitions-ports while externally they have to be connected to places.

In this approach port-mapping is a two-step procedure. Internally the ports of the interface of the internal net have to be mapped on to the ports of the macro-transition and externally the ports of the macro-transition have to be connected following the usual rules of Petri Nets. Within this paper in the sequel only hierarchical nets using formal macro-transition ports will be used.

Ports x and xs are input-ports, while S and L are output ports. X is of type **N** and the other ports of type [**N**] (list of naturals).

A transition t is called internal to transition t' if t is a node of a net instantiated by t' or by an internal transition of t'. Transition t' then is called an encloser of t and a minimal enclosure if there is no internal transition t'' of t' that is an enclosure of t. In example 4 transition $u : v$ is a minimal enclosure of transition a while transition a is an enclosure of transition a but not a minimal one. Two transitions t, t' are called mates if they have the same minimal enclosure (e.g. $u : v$ and c in the above example). A transition is called upper level transition if it has no enclosures. A place is called local if it is input or output of mates only (e.g. P_2 in example 4), it is called shared otherwise (e.g. P_3 in example 4). It should be mentioned that these definitions hold independently of whether there are formal macro-transition ports, or not.

Concerning firing rules the behaviour of Structured Petri Nets as described in section 5.7 is inherited.

6.5. Recursion

In our (informal) definition of hierarchical nets it has not been excluded that an internal transition t of a net n instantiates this net n. If done so, a recursive net structure is obtained. Such a structure, of course, is only legal if it is syntactically correct. i.e. the general rules of Petri Net graphs have to be respected and the edge-annotations have to be type-compatible with the places the edges are connected to. Graphically recursive instantiations are denoted by a slightly modified transition symbol:

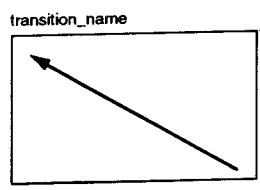

Fig. 6.11: Graphical representation of recursive macro-transition

The question now is whether the dynamic behaviour of recursive nets can be defined in a natural way. But this can be done easily just by combining the principals of coloured nets and hierarchical ones.

Whenever there is a recursive net definition we implicitly assume the tokens and the firing rules to be coloured. For convenience we will speak of recursion-path instead of colour in the sequel. The recursively defined macro-transition and each of its ocurrencies in the recursive definition have to have a unique identification.

The concatenation of these identifications in the case of recursive calls serves as a unique recursion path, identifiying for each token to which recursive activation it belongs. Initially the recursion-path is set to the identification of the entire macro-transition. When a subnet has to be initiated recursively, its input tokens are copied but with a recursion-path attached, which is the caller's firing path with the identification of the called occurence in the definition concatenated. If a recursively activated macro-transition is terminated, it marks its output-places (as usual) but with a recursion-path attached which is the actual internal recursion path with the last identification symbol truncated. Obviously only the usual techniques to handle recursion hve been expressed within the framework of hierarchical coloured nets.

A recursive definition, however, would be useless when not ancered. Therefore a macro-transition that is refined in a recursive way (obviously by definition recursion can occur only for macro-transitions) has to have two exclusive paths connected to the same input/output places and the predicates of the two paths have to be mutually invers. One path must not be refined recursively as it serves as the recursion's ancer. A simple example may serve to explain the principle:

Example 6:

The recursive Pr/T-Net of Fig. 6.12 performs Quicksort:

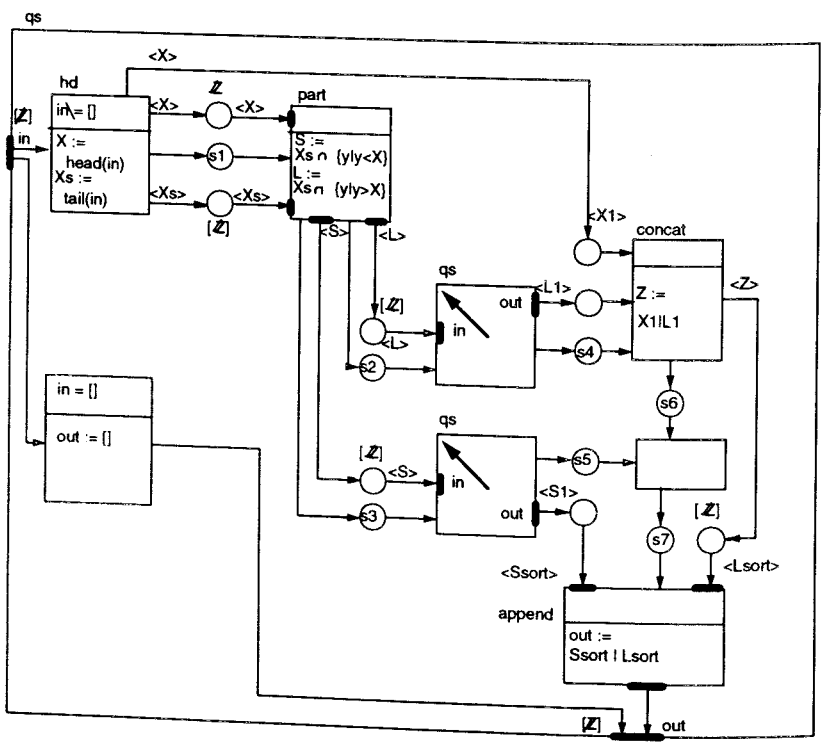

Fig. 6.12: Recursive definition of *quicksort*

Explanation:

The macro-transition *qs* takes a list of integers as input and produces a (sorted) list of (these) integers. The embedded transition *ancer* serves as ancer of the recursive definition: if the input-list is empty an empty output list is produced.

In all other cases first transition *hd* extracts the first element of the input list (x) and hands over the remaining list (xs) together with x to transition *part*. This transition partitions xs to a sublist (S) of elements that are smaller than x and another one (L) of elements that are larger than x. Both sublists are now sorted by recursively activating *qs*. $L1$ is the sorted list L, $S1$ the sorted list S. Transition *concat* forms the list z with x concatenated left to $L1$ while transition *appnd* concatenates the sorted sublists (now including x).

The entire control-flow is synchronized by places local to *qs* (note that by definition places connected to ports are shared ones) in such a way, that there is a sequence: *hd* then *part* then two activations of *qs* concurrently then (after termination of *concat* which is activated after the termination of the *qs* instance for sublist L) *append.*

The dynamic behaviour may be explained with the aid of a small example. For this discussion the places used for synchronisation purposes will be denoted by their identification used in the drawing ($S1$ to $S7$) while the value carrying places will be denoted by the variables attached to the input-arcs of transitions. Let's assume, that the list [4, 3, 5] has to be sorted. So in the (not drawn) input place connected to port *in* there is a token $[4,3,5]_0$ (we write here the recursion-path just as index of the tokens involved). As $[4,3,5]_0$ is not the empty list, transition *hd* fires, removing $[4,3,5]_0$ and producing $[4]_0$ in places x and $x1$, $[3,5]_0$ in place xs and a $token_0$ in $S1$. Now transition *part* is firable. By this, places x, xs and $S1$ become empty. Place S is marked with $[3]_0$, place L with $[5]_0$ and S_2, S_3 with a $token_0$ each. Now transitions qs_1 and qs_2 are firable. To make this discussion more readable, only qs_1 will be tracked. The token $[5]_0$ now is (virtually) copied to the input place of *qs* but with recursion-path 01, so there is now a token $[5]_{01}$. As this is not the empty list, *hd* fires, removing $[5]_{01}$, and producing $[5]_{01}$ in places $x, x1$ and $[]_{01}$ (the empty list) in place xs while $S1$ is marked with a $token_{01}$. Transition *part* demarks x, xs and $S1$, and marks L, S with $[]_{01}$ and $S2, S3$ with a $token_{01}$. Again let's track qs_1 only. Now by the recursive call we have a token $[]_{011}$ in the input place. This list being empty causes transition *ancer* to fire. It produces a token, which leaves the recursive macro-transition, i.e. a token $[]_{01}$ is produced in $L1$ and, as now qs_1 has fired for 01, $S4$ is marked with a $token_{01}$. But now transition *concat* can fire for 01-tokens. It removes $[5]_{01}$, $[]_{01}$ and the $token_{01}$ from its input places and produces a token $[5]_{01}$ in place *Lsorted* and a $token_{01}$ in S_6. As (by an analog procedure) qs_2 will produce a $token_{01}$ in S_5 and $[]_{01}$ in *Ssorted* transition *appnd* is now able to fire after S_7 has been marked with a $token_{01}$ (places S_5 and S_6 now are unmarked). By firing *appnd*, its input places are unmarked, and a token $[5]_{01}$ is produced. As this token leaves the definition of the recursive macro-transition it becomes a token $[5]_0$ which is placed in the output place of the respective occurrence of the transition, i.e. in $L1$. As there are now no more 01-tokens, *qs* fires for 0, producing a $token_0$ in S_4. Now transition *concat* can fire again. It now removes the token $[4]_0, [5]_0$ and $token_0$ from its input places and produces $[4,5]_0$ into *Lsorted* and a $token_0$ into S_6. As qs_2 will produce a token $[3]_0$ in Ssorted and a $token_0$ in S_5 finally (after firing of the intermediate synchronization transition) *appnd* can fire. It demarks *Ssorted*, *Lsorted* and S_7 and

produces a token $[3,4,5]_0$, which by leaving the macro-transition becomes an (uncoloured) token $[3,4,5]$. As there is now no more activity within *qs*, this transition fires, turning it to the passive state. It may now be reactivated externally with a new list to be sorted.

In the definition of the macro-transition *qs* (Fig. 6.12) there are two more macro-transitions included that need to be defined: *part* and *appnd*. These definitions are ommitted here.

6.6. Modelling with HR Pr/T-Nets

The above discussion has shown that hierarchical recursive Pr/T-Nets are a very powerful means to model complex systems in a convenient way. By their natural graphical representation this method supports the generation of specifications that are easy to understand. Readability is provided mainly by the very simple principle of Pr/T-Nets which inherited simplicity and, what is especially important for readability, locality from original Petri Nets but add the expressive power to specify both, control and data-operations. With the features added in this paper (hierarchy and recursion) all basic techniques to conquer complexity are present now. A complex system to be specified can now be decomposed into its basic components where each component is modelled as a macro-transition. By identifying the types of objects that such a macro-transition consumes and produces, the basic information streams are identified. Finally the concurrent overall control structure can be expressed using the usual Petri Net principles.

The hierarchy concept now allows macro-transitions to be refined. This may be carried out in a stepwise manner until an intended level of abstraction has been reached. Where adequate the refinement can be given by recursion. The refinement process may be divided between different implementation techniques: software where a scheduler takes over the part of the general control structure [RU93], hardware where the control structure can be mapped directly or any mixture (hardware/software codesign [GD92a, GD92b]). The modelling power of hierarchical recursive Pr/T-nets however is not limited to systems consisting of digital electronics together with software: the behaviour of arbitrary discrete systems can be modelled.

6.7. HR Pr/T-Nets and Flat Concurrent Prolog (FCP)

6.7.1. Introduction to FCP

In order to provide some information to understand the following discussion, some basic principles of Flat Concurrent Prolog (FCP) shall be discussed here. This section is more or less a short excerpt from [GLA92].

FCP is a general purpose logic programming language designed for concurrent programming and parallel execution. It was developed at the Weizmann Institute of Science in 1985 [Mi85]. The computational model of FCP is based on the process interpretation of logic programs [Sh86] in which the active parts of a computation are conceived as concurrent processes. These concurrent processes communicate via *shared logical variables*. An individual process is represented by a *goal* atom of the form $p(A_1, A_2, ..., A_k)$. By this a process of type p with arity k and the argument list $A_1, ..., A_k$ is identified. The $A_1, ..., A_k$ are arbitrary *logical terms*.

A process can perform one single operation, called *process reduction*. It is determined by the current values of the process' arguments when and how the process reduction can be executed.

Given a logic program P, the behaviour of a process $p(A_1, ..., A_k)$ is defined by the finite subset $C^{P/K} \subseteq$ of *program clauses* for predicate P and arity k. This subset $C^{P/K}$ is also called a *process procedure*.

Each clause $C_i^{P/K} \in C^{P/K}$ represents a rewrite rule for goal atoms of type P/K. It has the structure of a *guarded Horn clause* as defined below, where A, the G_i, and the B_j respectively represent atoms:

$$\underbrace{A}_{Head} \leftarrow \underbrace{G_1, G_2, ..., G_m}_{Guard} \mid \underbrace{B_1, B_2, ..., B_n}_{Body} \qquad (m, n \geq 0)$$

Guard predicates state conditions referring to process argument values. In order to perform a reduction on a process A' using some program clause $A \leftarrow G_1, ..., G_m \mid B_1, ..., B_m$, the goal atom A' has to unify with the clause's head atom A (as in ordinary Prolog) and in addition all guard predicates $G_1, ..., G_m$ must be fulfilled under the selected unification.

The *body part* $B_1, ..., B_n$ here is interpreted as a collection of atoms defining a multiset of concurrently active subprocesses. As a result of a successful reduction step, these subprocesses spawn a local subnetwork that replaces the reduced process. The global network state is updated due to the unification carried out. A process reduction operation is possible if the related process procedure contains at least one enabling clause (unifyable and fulfillable guards). Following the principle of OR-parallelism, all clauses of such a procedure are tried in parallel.

In the case of multiple applicable clauses a selection is made indeterministically under control of the *commit operator* '|' (guarded command indeterminacy) [Di75]. A clause is definitely chosen for reduction after its *commit operator* has been processed. All other concurrently regarded clauses for this reduction are discarded at this time. The commit operation is irreversible, i.e. there is no backtracking.

A clause may respawn a process of some type as the one it reduces. This special case will play an important role in the sequel. Another special case are unit clauses. They have the form $A \leftarrow G_1, ..., G_m \mid true$. The term *true* represents the empty process network. A unit clause results in process termination.

Up to now guard predicates of arbitrary complexity have been allowed. This results in languages that can't be executed efficiently. Therefore, so-called *flat languages* restrict the use of guard predicates to a predefined set of primitive and built-in operations. They can be evaluated immediately without performing subcomputations (processes) for guard predicates. In the execution of a flat language, the goal therefore corresponds to a flat collection of processes. FCP is just one (very powerful) example of flat languages. An overview on these languages is given in [Sh89].

The basic synchronization concept of concurrent logic programming is to delay reduction operations depending on the instantiation state of variables in the argument of processes. If argument variables of a process have not been sufficiently instantiated to be able to determine a reducing clause or to recognize that none of the potential clauses is applicable, this process is *suspended*. When process variables that caused a process suspension are instantiated, a repeated reduction is attempted.

FCP applies a reduction delaying mechanism based on the concept of *read-only* variables. In contrast to ordinary (*writable*) variables, there is no write access to read-only variables, except by instantiating their writable counterparts. A read-only variable belonging to a writable variable 'X' is denoted by '$X?$'. There may be more than one occurence of a '$X?$' for one variable 'X'. The writable variable 'X' and all

its read-only counterparts '$X?$' refer to the same physical writable instance. Any attempt to reduce a process using a clause that would affect a read-only variable within the process argument results in a process suspension.

In combination the variables X and $X?$ form a *communication channel* going from X to $X?$. However, in spite of the fact that, due to the single-assignment nature of logical variables, any simple variable can be instantiated at most once, a communication channel as described here allows the transfer of an infinite sequence of messages. For this purpose, the Prolog *list structure* is used. When the writing process is instantiating the writable variable, it does this by instantiating the variable with the actual message (*list head*) and an uninstantiated *list tail*.

6.7.2. Basic Approach

Here only the problem of executing HR Pr/T-Nets using FCP (i.e. translation to FCP) shall be investigated. When examining the two models, similarities but also differences can be observed:

- A transition in Pr/T-Nets has to perform two steps to decide whether it is firable:
  - First it has to check whether there are sufficient tokens of the right kind in its input places so that a valid interpretation is obtained.
  - Secondly the transition's predicate has to be checked against such an interpretation.

  At each point of time there may be no, exactly one or more valid interpretations. If there are valid interpretations the predicate may be true for no, exactly one, or more valid interpretations. In the latter case the transition is firable multiple times.

  A FCP clause head has to be unified with logic variables. There may be no, exactly one, or more possible unifications. If unification is successful the (optional) guard is checked. For a given unification this guard predicate may be true or not. If the guard is true, the clause's body is executed irreversibly. So the two concepts are somewhat similar. The main difference occurs if there is more than one unification with the guard predicate being true. Here FCP selects indeterministicly just one alternative while in HR Pr/T-Nets a transition will fire for all valid interpretations. In order to solve simpler cases first, in the first instance it is assumed, that there is never more than one token in a place. In this case a place can be modelled by a logic variable and the corresponding input arc by a corresponding read-only variable.

- Pr/T-Nets have a static structure, i.e. the entire net exists statically and only the token flow introduces a dynamic concept. In contrast to this, FCP has a dynamic structure. Processes are created only on request, i.e. if they are contained in the body of a clause that has successfully been unified and the guard of which is true. In addition a FCP process disappears as soon as its task is finished. An everlasting process, however can be modelled in FCP by the following schema:

$$\begin{aligned} transition - name([In1? \mid In1s?], ..., [Ink? \mid Inks?], \\ [Out1 \mid Out1s], ..., [Outn \mid Outns]) : - \\ transform(In1?, In2?, ..., Ink?, Out1, Out2, ..., Outn) \end{aligned}$$

$transition - name([In1s?], ..., [Inks?],$
$[Out1s], ..., [Outns]).$

Notice that now lists (i.e. sequences) of variables are processed. This models one-to-one the sequence of tokens flowing through a net. The appearance of a token in a place now is modelled by a variable being initiated which enables the process *transition-name*. It performs the calculation of new tokens and after this spawns itself. By the structure of the arguments the clause cannot be unified with empty input lists, i.e. it is activated only if each of the read-only-lists carry an initiated head element. Arguments of the process *transform* whose values don't influence the values of the token produced can be omitted of course. Such tokens are just "consumed" by the respawning clause. An entire Pr/T-Net (up to now without hierarchy and recursion) is now modelled by the schema:

net:-
$trans_1$ $([In11? \mid In11s?], ..., [In1k_1 ? \mid In1k_1 s?],$
$[Out11 \mid Out11s], ..., [Out1n_1 \mid Out1n_1 s]),$
$\vdots$
$trans_m$ $([Inm1 ? \mid Inm1s?], ... [Inmk_m ? \mid Inmk_m s?],$
$[Outm1 \mid Outm1s], ..., [Outmn_m \mid Outmn_m s]).$

where the different $trans_i$ clauses (processes) are defined as above. Places shared by different transitions are modelled by variable-identifiers shared by processes. Some of the lists have to be provided with instantiated heads. This models the initial marking and allows some processes to be activated.

- Pr/T-Nets like any kind of Petri Nets consume tokens when firing. This is modelled in the above schema by discarding the head-elements of each argument-list when a transition spawns itself. However, by this method only the "consuming" process gets notice of the "token" being consumed. So, this method works only with places without forward conflict. As, of course, nets with conflicts have to be modelled as well, a special solution has to be found. Fortunately this problem has a relatively simple solution as long as a conflict is decided deterministicly based on attached predicates or global markings, i.e. whenever there is a conflict, at most one of the respective transitions really can fire. Assume transitions $t_1, ..., t_m$ being in conflict w.r.t. place P_1. For each of these transitions t_i there is an additional variable $[Remove\_token_i \mid Remove\_tokens_i]$. The variable $Remove\_token_i$ is instantiated during the firing process of $transition_i$. On the other hand all $[Remove\_token_j \mid Remove\_tokens_j]$ with $j \neq i$ are included in the parameter list of each $transition_i$ as r/o-variables. The "normal" clause for $transition_i$ is augmented by guard predicates $unknown\ (Remove\_token_j)$ for all $j \neq i$ and for each such j a clause is added of the form:

$transition_i([In1? \mid In1s?], ..., [Ink? \mid Inks?],$
$[Out1 \mid Out1s], ..., [Outn \mid Outns],$
$[Remove\_token_1? \mid Remove\_tokens_1?], ...,$
$[Remove\_token_m? \mid Remove\_token_m s?],$
$[Remove\_token_i \mid Remove\_token_i s]) : -$
$unknown(Remove\_token_1?), ..., unknown(Remove\_token_m?) \mid$
$transform(..., remove\_token),$
$transition_i(In1s?, ..., Inks?, Out1s, ..., Outps,$

$Remove\_token_1?, ..., Remove\_tokens_m?, Remove\_tokens_i).$

$transition_i(...) : -known(Remove\_token_1) \quad | \quad transition_i(...).$

$\vdots$

$transition_i(...) : -known(Remove\_token_m) \quad | \quad transition_i(...)$

- Pr/T-Nets allow situations where transitions use a place for both, input and output. The semantics of FCP however require the set of r/o variables to be disjoint of that of writeable variables. If a transition just leads a token unchanged in such an input/output place ("test-only places") the place can simply be ommitted in the list of output places and in addition the first list-element of the respective r/o variable has not to be removed when respawning the procedure during "normal" firing. However, it has to be removed when a "remove-token"-message is received from a conflicting transition as explained above. A transition handled in this way must not be considered as an active competitor in a conflict situation. As it does not really consume a token it is also not allowed to send a "remove-token"-command.

  If the transition, however, changes a token in an input/output-place, another technique has to be used. In this case, the transition does not fire directly into its output place but into an auxiliary one. An additional transition, let it be called "token-mover", then fires the token into the final destination. So if it is assumed, that transition t uses place p as input/output place, the following skeleton is obtained:

  $t(..., [P? \mid Ps?], ..., [Paux \mid Pauxs], ...),$

  $token\_mover([Paux? \mid Pauxs?], [P, Ps]),$

  where t and $token\_mover$ are defined in the usual way.

6.7.3. Handling of Hierarchy

FCP is hierarchical by nature. This makes modelling of hierarchical Pr/T-nets easier. The main problem to be addressed is that a macro transition completes its firing not earlier than its embedded net becomes dead. It has also to be modelled that a macro transition always starts from a fixed initial internal marking. The basic idea to handle this correctly, is to introduce a process, that decides whether all processes modelling the embedded net of a macro transition are suspended, indicating that the net is dead. In this case these processes have to be aborted and the firing of the macro transition has to be completed. The skeleton of a FCP model for a macro transition looks like the following:

Assume a macro transition mt with input places $ip1, ..., ipn$, output places $op1, ..., opn$ and a predicate $guard$. This is modelled by:

$mt([Ip1? \mid Ip1s?], ..., [Ipn? \mid Ipns?], [Op1 \mid Op1s], ..., [Opm \mid Opms]) : -$
$guard \mid t1(..., Abort?),$

$\vdots$

$tk(..., Abort?),$
$transform\_mtx(Ip1? \mid Ip1s], ..., [Ipms?],$
$[Op1 \mid Op1s], ..., [Opm \mid Opms], Abort?),$
$termchk(T1?, ..., Tk?, Abort).$

Each embedded transition t_i is translated to

$ti(..., Abort?) : -$
$\quad unknown(Abort?) \mid transform\_ti(...),$
$\quad\quad ti(..., Abort).$
$ti(..., Abort?) : -$
$\quad known(Abort?) \mid true$

Respawning *mt* now becomes part of *transform_mtx*, to make sure that it takes place only after it has finally fired:

$transform\_mtx([Ip1? \mid Ip1s], ..., [Ipm? \mid Ipms?],$
$\quad\quad [Op1 \mid Op1s], ..., [Opm \mid Opms], Abort?) : -$
$known(Abort?) \mid transform\_mt(Ip1?, ..., Ipm?, Op1, ...Opm),$
$\quad mt(Ip1s?, ..., Ipms?, Op1, ...Opm).$

6.7.4. Handling of Coloured Nets

Up to now it has been assumed that no more than one token is contained in a place at any time. This assumption makes no sense in coloured nets. As already discussed, coloured nets are no real extension of Pr/T-Nets. If colours are present, this means nothing else other than adding one component "colour" to the datastructure that describes tokens. The existence of multiple tokens can be modelled by using lists of lists instead of simple lists to model places. The check for coloured firability and the coloured firing can then be modelled using standard techniques.

As coloured nets have been introduced here only to support the definition of recursive nets and as this special case can be handled in a more elegant manner, coloured nets are not discussed further.

6.7.5. Handling of Recursion

FCP is recursive by nature. So the mechanisms introduced to define recursive Pr/T-Nets are implicitly part of the FCP language. Recursive Pr/T-Nets therefore translate directly to FCP. However, some details have to be considered. First of all, a recursive macro transition plays two roles: that of the macro transition itself and that of an embedded transition. This causes problems with the abortion mechanism. Fortunately a recursive transition has to be aborted exactly after its ancer-definition has been executed. Modelling the ancer-definition as alternative clause for the recursive macro-transition without respawning solves the problem in the case of a recursive call. However, the macro-transition might be called from the outside with the same parameter that causes the recursion ancer to be selected. In this case a respawning has to take place. This problem is solved by introducing an additional level of (FCP-)hierarchy. A recursive macro transition spawns an "internal" version of itself and upon termination of this "internal"version ($\hat{=}$ termination of recursion) respawns itself. It is the "internal" version of the macro transition that is called recursively. There is no respawning included in the definition of this "internal" version other than that caused by recursive calls. These recursive calls are terminated by selecting the ancer-clause. As recursive macro-transitions have to be completely dynamic, each transition in the body of such a recursive macro transition has to disappear immediately after firing. Therefore the respawning part of the definition of transitions has to be ommitted in this case. By the same reason, the termination detection needed otherwise for macro-transitions, is obsolete in the recursive situation.

The quicksort-example shown in Fig. 6.12 may serve to illustrate the approach. To make the code more readable, two additional simple transformations are applied:

- the operation of transitions *hd* and *concat* are simple list operations that can be made part of the unification procedure. Therefore they are ommitted and the clause-heads of *qs* and *appnd* are rewritten to take over this task.
- the sequencing caused by markings of places $S_1, S_2, S_3, S_4, S_5, S_6, S_7$ are redundant in this case. They are present only by syntactical reasons as macro-transitions have to be activated explicitly. In the FCP-version they are ommitted.

With these simple transformations the following FCP-code for quicksort is obtained:

```
1  qs ( [ [ Inh ? | Inhs ? ] | Ins ? ], [ Start ? | Starts ? ],
     [ Out | Outs ], [ Ready | Readys ] ) :-
     qsi ( [ Inh ? | Inhs ? ], Out, Iready ),
     transform_qsx ( [ [ Inh ? | Inhs ? ] | Ins ? ],
     [ Start ? | Starts ? ],
          [ Out | Outs ], [ Ready | Readys ], Iready ? ).
2  transform_qsx ( . . . ) :-
     transform_qs ( Out, ready ),
     qs ( Ins ?, Starts ?, Outs, Readys ).
3  qsi ( [ Inh ? | Inhs ? ], Out, Iready ) :-
     qsi ( [ Inh ? | Inhs ? ], Out, Iready ) :-
     part ( Inh ?, Inhs ?, s, e ),
     qsi ( E ?, E1, Iready1 ),
     qsi ( S ?, S1, Iready2 ),
     appnd ( S1 ?, [ Inh ? | E1 ], Out ).
4  qsi ( [ ], [ ], iready ).
```

Explanation:

Clause 1 defines *qs*. This process takes upon activation *(Start?)* a list of integers *([Inh? | Inhs?])*, produces a sorted list of integers *(Out)* and reports to have finished the job (*Ready*). To do the job, *qs* spawns an internal version of itself with the actual parameters only but an additional parameter (*Iready*) that reports that the internal version has been finished. After this *qs* performs its firing as defined by clause 2. Place *Out* is marked by the sorted list as obtained from *qsi* and place *Ready* is marked by the (arbitrary) value *ready*. In addition *qs* is respawned in the usual way with the heads of the parameter lists truncated

Clause 3 defines the (non-ancer version of) *qsi*. It can be seen that the task of transition *hd* is taken over by making visible the list-structure in the head of *qsi* and using its components in the head of *part*. Similarily the task of transition *concat* is included in the head of *appnd*. Clause 4 defines the ancer-version. This clause also instantiates *Iready* to the (arbitrary) value *iready*.

The colouring mechanism described in section 5 is now carried out implicitly by the FCP runtime system. The FCP version of the recursive macro transitions *part* and *appnd* is obtained in a similar manner.

7. System Level Design Activities

7.1. System Level Simulation

By the heterogeneity of typical system level models, monolithic simulation systems hardly seem to be adequate. Multisimulator-systems may therefore be the best solution. Three major problems have to be solved when dedicated simulators have to be coupled to a multisimulator:

- data exchange between the different simulators,
- synchronization of the individual simulators,
- a unified user interface.

While data exchange can be handled fairly easily by defining universal exchange formats and eventually adding information about transmitter details, the synchronization turns out to be the central problem. The simulators involved have to be kept at least synchronous enough so that at each point of time a data exchange occurs, this point of time is legal for all simulators involved. There are two extreme solutions for this problem:

- the ("oversynchronizing") supervisor approach,
- the ("undersynchronizing") time warp method.

The supervisor approach being a "pessimistic" one always allows only the simulator that plans to let the event happen in the closest global future perform this action. The method is safe but doesn't allow any concurrency. Time warping [JS85] "optimistically"assumes that the simulators can run completely independently. This assumption is OK as long as no data is transmitted to the local past of a simulator. If so, the receiving simulator has to be rolled back and resumed at an earlier local point of time. Of course when doing this, all messages sent to other simulators in the meantime have to be canceled by sending "antimessages". This may cause additional rollbacks at the receivers of these antimessages. Recently very powerful multisimulator frameworks have been reported, SICS [NI91, OC91] being an especially promising system. Currently CFI is working at a standard for the coupling of heterogenous simulators. This standard on a "simulator backplane" will be similar to the SICS-approach which heavily influenced this standard.

Simulation at system level is mainly used to offer a workbench to the system engineer to perform experiments. Therefore special support to plan, perform, and analyse experiments has to be offered. For this purpose AI techniques have been proposed by a couple of authors [AP86, EO86, ER88, LA86, SM85]. With such techniques a knowledge based automated experimenter that performs experiments in a goal-oriented manner can be implemented. A very advanced system for this purpose has been designed and implemented by H. Pfaffhausen [PF91]. Such a system is a step to closing the gap between simulation and "formal" methods.

7.2. System Level Analysis and Partitioning

Besides experimenting with a modelled system various kinds of analysis may be carried out. This analysis can be based on simulation results or statically on the model itself. Performance analysis is the area where most results have been achieved. By

applying queuing theory, simpler situations can be solved analytically while in other cases simulation has to be applied. In this case not the system's functionality is simulated but the behaviour at the assumed queues. HIT [BE86] is an excellent example of such a performance analysis tool applicable to system level. It models systems in a very clean client/server way and offers hierarchy to cope with complex systems. Testability analysis, well understood at lower levels within the electronics domain is rarely supported at system level. Rule based approaches like that described in [BI91] may be promising. Similar approaches may serve for an analysis of fabricability and maintainability. In order to support a clever partitioning, analysis of similarity with predefined objects, but also within a description, is helpful. A partition where, relatively, many parts can be mapped into few predefined classes is a good candidate for an economic one. Feature oriented retrieval operations on libraries of rather complex objects are therefore needed. At this level of abstraction it makes nearly no difference whether the "similar" object is similar because of offering the required methods directly as hardware or by software solutions. Therefore such a retrieval system is equally helpful to search in libraries for OO programming and in libraries of hardware components. Designing such retrieval systems will be one of the most challenging tasks in system engineering of the near future.

8. System Level Design and Concurrent Engineering

When system level design is carried out all aspects of a product's lifecycle have to be considered. This coherent consideration of multiple interdependencies is called Concurrent Engineering (CE) [SS91]. Aspects to be considered include system functionality, performance, price, price/performance ratio, maintainability, fabricability, marketing aspects, substitutability of preproducts, management cost, recycling costs, legal aspects, etc.

Concurrent Engineering has been defined by the US Institute for Defense Analysis as:

"Concurrent Engineering is a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. This approach is intended to cause the developer, from the outset, to consider all elements of the product life cycle from conception through disposal, including quality, cost, schedule and user requirements".

All these aspects have a very limited view of the entire problem and try to find local optima. To do so, influences of other aspects have to be considered, not necessarily knowing the inside of the decision-making process of the influencing aspects. Concurrent engineering has to provide the necessary information to the individual aspect-processing agents continuously and, based on a global engineering model, tries to find a global optimum as a compromise of the local optima provided. A concurrent engineering model therefore has the same similarities as a system level model of an object to be designed. System level design methods therefore may influence concurrent engineering techniques. Even similar computer support seems to be probable. Two main areas for computer support of CE can be identified: support of concurrency and support to keep track of the various, independent aspects of engineering. The neccessity to support various design teams results in a strong demand on distributed processing and team-work tools. An elaborate CE environment will include all neccessary services to support high speed LANs, MANs and WANs including multimedia and videoconferencing. On top of these base technologies Computer Supported Cooperative Work (CSCW) systems have to be implemented. Client-server architectures seem the appropriate basic approach to grant these de-

mands. OSF's Distributed Computing Environment (DCE) is an example of an architecture supported by industry.

In any case a powerful design framework is needed for system level design with or without connected concurrent engineering. When embedded in a CE environment this framework aspect becomes even more evident. Framework architectures have been standardized by the CAD Framework Initiative (CFI) a worldwide association of about 50 institutions, most case companies active in the electronics CAD market. In the CFI framework model, there exist tool slots, where various tools may be integrated as long as they are in compliance with the CFI standards on interfaces. This integration procedure has to be considered in three different and orthogonal views:

- data integration
- control integration
- user interface integration

This includes additional basic components of a CFI compatible framework:

- a methodology management,
- a user interface management,
- a common design representation based on data management.

All these components are based on a standardized system environment and supported by standardized services for intertool communication. Based on such a framework, complex design environments can be built fairly easily [BKR93].

The core component of such a framework is an object oriented data base management systems (OMS) [FF91]. This OMS has to handle objects of different complexity and efficiently retrieve objects in a feature oriented way. In addition a powerful event handling and trigger mechanism has to be provided to inform and trigger tools that are working concurrently at different aspects of designing a specific system. This is the base service for CE.

Another important service of such a framework is design flow management. [KU92, Za92, LJD92] It manages all the versions of obtained design documents, application of tools in a concurrent or sequential manner, makes decisions about tools being best suited for a specific design, produces the reports that make the design process understandable and controllable. The third basic component of such a framework is a unified user interface management system (UIMS) helping to implement ergonomic user interfaces. Design frameworks have been identified to be the central service to be provided for concurrent engineering. They are investigated scientifically [RA87, RW91, NR92] and commercial products are emerging. One of the most advanced solutions for the framework problem is the already mentioned JESSI Common Framework (JCF) [ST92].

Currently there is an emerging trend towards so called "Federated Systems". This is an approach to keep existing frameworks or comparable systems alive and to organize the neccessary interaction. Such an approach allows the saving of the enormeous investments such companies have spent in Product Information Management Systems (PIMS), CAD Frameworks, CIM environments etc., and at the same time the obtaining of the integration needed to achieve Concurrent Engineering. The design of federated systems is made easier by the fact, that at least PIMS and CFI compatible CAD frameworks have a similar architecture.

9. Conclusion

More and more system level design turns to the next challenge of engineering. After a long tradition of departmentalisation (at least within the western culture) where not only engineering has been looked at separately from other professional activities but also even withing engineering the different disciplines have been divided into fairly independent subactivities, new structures have to evolve. These new structures have to tackle the entire life cycle of products as a whole and at the same time activities have to be organized in as concurrent a way as possible.

A major problem to be solved in this context is to construct a common modelling base. Such a modelling base has to be applicable not only to technical objects but also to organizational ones like (e.g.) design processes. In this contribution the attempt has been made to propose extended Pr/T-Nets as base model. Such a model is intended to be used mostly as an internal conceptual model. The interface with human users should be carried out in a multiparadigmatic way, i.e. the computer system has to adapt to the user's preferences and demands and not vice versa. Some of the numerous possible external conceptual models have been discussed in the beginning of this contribution.

The modelling aspects also constitute a basis for concurrent engineering, the art of engineering which seems to be the most promising one to overcome today's organizational deficiencies.

References

[AH90] J.K. Annot, P.A.M. de Haan: POOL and DOOM: the Object Oriented Approach. P.C. Treleaven (E.): Parallel Computers, Object-Oriented, Functional, Logic. Wiley, 1990

[AG88] L.M. Augustin, B.A. Gennart, Y. Huh: Verification of VHDL Design Using VAL. Proc. 25th DAC, 1988

[AP86] H.H. Adelsberger, U.W. Pooch et al: Rule Based Object Oriented Simulation Systems. In [LA86]

[BE86] H. Beilner: Workload Characterization and Performance Modelling Tools. G. Serazzi (Ed.): Workload Characterization of Computer Systems & Computer Networks. North Holland, 1986

[BI91] M. Bidjan-Irani: A Rule-Based Design-for-Testability Rule Checker. IEEE Design & Test of Computers, March 1991

[BK91] J. Bartolazzi, K. Kirsch, K. Neusinger, K.D. Müller-Glaser: Towards an Integrated Environment for Microsystem Design. F.J. Rammig, R. Waxmann (Ed.): Electronic Design Automation Frameworks, North Holland, 1991

[BKR93] M. Brielmann, E. Kupitz, S. Rudolph: Hardware Engineering Environment. in: P. Schefstrm, G. van den Brock (Eds.): Tool Integration-Environments and Frameworks, Wiley, 1993

[BO81] D. Borrione: Langue de description des systemes logiques - Proposition pour une methode formelle de definition. These d'Etat, INPG Grenoble, 1981

[**BOP89**] H.J. Burckhardt, P. Ochsenschlger, R. Prinoth: Product Nets - A Formal Description Technique for Cooperating Systems, GMD Studien Nr. 165, Gesellschaft f. Mathematik u. Datenverarbeitung mbH, St. Augustin, 1989

[**BR81**] R.E. Bryant: MOSSIM: A Switch Level Simulator for MOS-LSI. Proc. 18th DAC, 1981

[**CC88**] CCITT Recommendation Z.100: Specification and Description Language SDL. AP IX-35, 1988

[**CG84**] K.L. Clark, S. Gregory: PARLOG: Parallel Programming in Logic. Imperial College, London, Research Report DOC 84/4, 1984

[**CH79**] Y. Chu: Introducing CDL. IEEE Computer, Dec. 1979

[**CK81**] L.A. Cherkasova, V.E. Kotov: Structured Nets. Proc. MFCS'81, Springer LNCS 118, 1981

[**CM81**]W.F. Clocksin, C.S. Mellish: Programming in PROLOG. Springer, 1981

[**DE78**] T. DeMarco: Structured Analysis and Systems Specification. Prentice Hall, 1978

[**DA87**] DACAPO III System User Manual. Dosis GmbH, Dortmund

[**DD75**] J.R. Duley, D.L. Dietmeyer: A Digital System Design Language (DDL). IEEE ToC, C-24, No. 2, 1975

[**EH92**] W. Ecker, M. Hofmeister: The Design Cube - A New Model for VHDL Designflow Representation. Proc. Euro-DAC'92, 1992

[**EO86**] M.S. Elzas, T.I. Ören, B.P. Zeigler: Modelling and Simulation Methodology in the Artificial Intellegence Era. North Holland, 1986

[**ER88**] H.W. Egdorf, D.D. Robert: Discrete Event Simulation Methodology in the Artificial Intellegence Environment. Proc. Conference on AI and Simulation, AI Papers, 1988

[**FF91**] W. Fox, J. Friedrich, R. Hopp, T. Kathöfer, A. Meckenstock, D. Nolte, K. Pielsticker, G. Reitmeyer, F. Rupprecht, M. Schrewe: The Architecture of the Object Management System within Cadlab Framework. F.J. Rammig, R. Waxmann (Ed.): Electronic Design Automation Frameworks, North Holland, 1991

[**FT87**] I. Foster, S. Taylor: Flat Parlog: A Basis for Comparison. Int. Journal of Parallel Programming 16.2,1987

[**GA87**] D.D. Gajski: The Structure of a Silicon Compiler. Proc. of IEEE ICCD, 1987

[**GD92a**] R.K. Gupta, G.DeMicheli: System Synthesis via Hardware-Software Codesign. CSL Technical Report CSL-TR, Stanfort University, 1992

[**GD92b**] R.K. Gupta, G. DeMicheli: System Level Synthesis Using Re-Programmable Components. Proc. EDAC'92, 1992

[GH78] J. Guttag, J.J. Horning: The Algebraic Specification of Abstract Data Types. Acta Informatica, 10, 1978

[GH91] U. Glässer, G. Hannesen, M. Kärcher, G. Lehrenfeld: A Distributed Implementation of Flat Concurrent Prolog on Multi-Processor Environments. Proc. First International Conference of the Austrian Center for Parallel Computation, 1991

[GK90] U. Glässer, M. Kärcher, G. Lehrenfeld, N. Vieth: Flat Concurrent Prolog on Transputers. Journal of Microcomputer Applications, Academic Press, Vol. 13, No.1, 1990

[GL81] H.J. Genrich, K. Lautenbach: System Modelling with High-Level Petri Nets. Theoretical Computer Science, 13, 1981

[GL92] U. Glsser: A Distributed Implementation of Flat Concurrent Prolog on Multi-Transputer Environments. in: P. Kacsuk, M.J. Wise (Eds.): Implementations of Distributed Prolog, Wiley, 1992

[GU90] W. Glunz, G. Umbreit: VHDL for High-Level Synthesis of Digital Systems. Proc. 1st European Conference on VHDL, 1990

[GV91] W. Glunz, G. Venzl: Hardware Design Using CASE Tools. Proc. IFIP VLSI'91, 1991

[HA77] R. Hartenstein: Fundamentals of Structured Hardware Design. North Holland, 1977

[HA87] D. Harel: Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8, 1987

[HE90] G. Held (Ed.): Sprachbeschreibung GRAPES. Siemens AG, 1990

[HI85] P.N. Hilfinger: A High-Level Language and Silicon Compiler for Digital Signal Processing. Proc. IEEE Custom Integrated Circuits Conf., 1985

[IMT87] N. Ichiyoski, T. Miyazaki, K. Taki: A Distributed Implementation of Flat GHC on the Multi-PSI. in: J.L. Lassez (Ed.): Logic Programming-Proceeding of the Fourth Int. Conference on Logic Programming, MIT Press, 1987

[JE91] K. Jensen: Coloured Petri Nets: A High Level Language for System Design and Analysis. in: G. Rozenberg (Ed.): Advances in Petri Nets, Springer Lecture Notes in Computer Science, No. 485, 1991

[JS85] D.R. Jefferson, H.A. Sowizral: Fast concurrent simulation using the time warp mechanism. Proc. SCS Distributed Simulation Conference, 1985

[KA90] T.H. Krodel, K.J. Antreich: An Accurate Model for Ambiguity Delay Simulation, Proc. EDAC'90, 1990

[Ki92] D. Kirstein: TransNet - Ein Interpreter zur mengentheoretischen Transformation hierarchischer Petrinetze. Diplomarbeit, Univ.-GH Paderborn, FB 17, 1992

[**KK91**] B. Kleinjohann, E. Kupitz: Tight Integration in a Hardware Synthesis System. In [RW91]

[**Ku92**] E. Kupitz: Design Assistance in Concurrent Integrated Environments. Proc. 3rd IFIP Workshop on Electronic Design Auto-mation Frame-works-(EDAF'92), North Holland, 1992

[**LA86**] P.A. Luker, H.H. Adelsberger: Intelligent Simulation Environments. SCS Simulation Series, 17:1, 1986

[**LJD92**] J.C. Lopez, M.F. Jacome, S.W. Director: Unifying Tool, Data and Process Flow Management. Proc. Euro-DAC, 1992

[**LGR92**] B. Lutter, W. Glunz, F.J. Rammig: Using VHDL for Simulation of SDL Specifications. Proc. Euro-DAC'92, 1992

[**LM92**] C. Le Faou, J. Mermet: Introducing CASCADE, Control Graph in VHDL. in: J. Mermet (Ed.): VHDL for Simulation, Synthesis and Formal Proofs of Hardware, Kluwer, 1992

[**LR83**] K.D. Lewke, F.J. Rammig: Description and Simulation of MOS Devices in Register Transfer Languages. Proc. IFIP VLSI 83, North Holland, 1983

[**MB87**] B. Moller-Pedersen, D. Belones: Rational and Tutorial OSDL: An Object-Oriented Extension of SDL. Computer Networks and ISDN Systems 13, 1987

[**ME70**] J. Mermet: Définition du Langage CASSANDRE, Thèse Doctorat-ingénieur, Grenoble, 30 mars 1970

[**MI85**] C. Mierowskiy, S. Taylor, E. Shapiro, J. Levi, S. Safra: The Design and Implementation of Flat Concurrent Prolog. Technical Report CS 85-9, Dept. of CS, The Weizmann Institute of Science, Rehovot, Israel, 1985

[**MR89**] W. Müller, F.J. Rammig: ODICE: Object Oriented Hardware Description in CAD Environment. Proc. IFIP CHDL 89, North Holland, 1989

[**MT90**] R. Milner, M. Tofte, R. Harper: The Definition of Standard ML. MIT Press, 1990

[**NA89**] C. Nagel: Generierung funktionaler Modelle für v.Neumann Rechnerarchitekturen. Diplomarbeit, Univ.-GH-Paderborn, FB 17, 1989

[**Na92**] K. Nakajima: Distributed Implementation of KL1 on the Multi-PSI. in: P. Kacsuk, M.J. Wise (Eds.): Implementations of Distributed Prolog, Wiley, 1992

[**NI91**] M. Niemeyer: Simulation of Heterogeneous Models With a Simulator Coupling System. Proc. SCS 1991 European Simulation Multiconference, Juni 1991

[**NR92**] M. Newman, (ed.), Tom Rhyne: Electronic design automation Frameworks: when will the promise be realized? Proceedings of the 3rd IFIP WG 10.2/WG 10.5 Workshop on Electronic Design Automation Frameworks, 1992

[OC90] A. Oczko: Hardware Design with VHDL at a very high level of abstraction. Proc. 1st European Conference on VHDL, 1990

[OC91] A. Oczko, Ch. Oczko: Putting Different Simulation Models Together - The Simulation Configuration Language VHDL/S. Proc. IFIP CHDL 91, North Holland, 1991

[PB83] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly: CONLAN Report. Lecture Notes in Computer Science No. 151, Springer, 1983

[PF91] H. Pfaffhausen: Ein wissensbasierter Ansatz zur automatischen Durchführung von Experimenten in der Logiksimulation. Dissertation, Universität-GH-Paderborn, 1991

[PS90] B. Plorin, M. Schweins: Dialogorientierte Generierung von Mikroprozessormodellen. Diplomarbeit, Universität-GH-Paderborn, FB 17, 1990

[Ra80a] Five Valued Quasi Real Boolean Functions, Proc. 5th. European Meeting of Cybernetics and System Research, 1980

[Ra80b] F.J. Rammig: Structured Parallel Programming with a Highly Concurrent Programming Language. in: Atti di Congresso Annuale AICA'80, 1980

[RA87] F.J. Rammig (Ed.): Tool Integration and Design Environments. North Holland, 1987

[RA89] F.J. Rammig: Systematischer Entwurf digitaler Systeme. B.G. Teubner, 1985

[Ra92] F.J. Rammig: Synthesis Related Aspects of Simulation. in: P. Michel, U. Lauther, P. Duzy (Eds.): The Synthesis Approach to Digital System Design, Kluwer, 1992

[Re85] W. Reisig: Petri Nets: An Introduction. Springer, 1985

[ROS77] D.T. Ross: Structured Analysis (SA): A Language for Communicating Ideas. in: IEEE ToSE SE-3:1 (1977)

[RU93] M. Rupprecht: Implementierung und parallele Verarbeitung von Kommunikationssoftware. Teubner Texte zur Informatik, Band, 1993

[RW91] F.J. Rammig, R. Waxmann (Eds.): Electronic Design Automation Frameworks. North Holland, 1991

[SDL92] CCITT: CCITT Specification and Description Language SDL, Recommendation Z. 100 (SDL'92). Genf 1992

[SE90] M. Seutter (Ed.): Glass: A system description language and its environment, Introduction and User manuals. University of Nijmegen, NL, 1990

[SH86] E. Shapiro: Concurrent Prolog: A Progress Report, IEEE Computer 19,8, 1986

[SH87] E. Shapiro: Concurrent Prolog: Collected Papers, Vol. 2, MIT Press, 1987

[**SH89**] E. Shapiro: The Family of Concurrent Logic Programming Languages. ACM Computing Surveys 21,3, 1989

[**SM85**] R.E. Shannon, R. Mayer, H.H. Adelsberger: Expert Systems and Simulation. SIMULATION, Vol. 44, Juni 1985

[**SP92**] J.M. Spivey: The Z Notation, A Reference Manual, 2nd Edition, Prentice Hall, 1992

[**SS91**] R.A. Sprague, K.J. Singh, R.T. Wood: Concurrent Engineering in Product Development. IEEE Design & Test of Computers, March 1991

[**ST86**] B. Stroustrup: The C++ Programming Language, Addison-Wesley, 1986

[**ST92**] B. Steinmller: The JESSI-COMMON-FRAME-Project - A Project Overview. in: NR92

[**SU85**] N. Suzuki: Concurrent Prolog as an Efficient VLSI Design Language. IEEE Computer, Vol. 18, No. 2, 1985

[**SU90**] U. Suffrian: Vergleichende Untersuchungen von State-Charts und strukturierten Petri-Netzen. Dipl.Arb., Univ.-GH-Paderborn, FB 17, 1990

[**TB84**] Teledyne Brown Engineering: IORL Reference Manual. Huntsville, AL, 1984

[**TH91**] D. Thomas: The Verilog Hardware Description Language. Kluwer, 1991

[**TLK90**] T. Tikkanen, T. Leppnen, J. Kivel: Structured Analysis and VHDL in Embedded Asic Design and Verification, Proc. EDAC'90, 1990

[**UE85**] K. Ueda: Guarded Horn Clauses. ICOT Techn. Report TR-103, Tokyo, 1985

[**VH87**] IEEE Standard VHDL Language Reference Manual. IEEE IStd 1076, 1987

[**VN91**] F. Vahid S. Narayan and D.D. Gajski: SpecCharts: A Language for System Level Synthesis. Proc. IFIP CHDL 91, North Holland, 1991

[**WM85**] P.T. Ward, S.J. Mellor: Structured Development for Real-Time Systems, vol. 1-3, Yourdon Press, N.Y. 1985

[**WS87**] D. Weinbaum, E. Shapiro: Hardware Description and Simulation Using Concurrent Prolog. Proc. IFIP CHDL 87, North Holland, 1987

[**YC79**] E. Yourdan, L. Constantin: Structured Design: Fundamentals of a Discipline of Computer Program and Design. Prentice Hall, 1979

[**Za92**] M. Zanella: Principles of Design Methodology Management for Electronic CAD Frameworks. Proc. EDAC'92, 1992