Approaching System Level Design

FRANZ J. RAMMIG
Universität-GH-Paderborn, FB Mathematik/Informatik,
D – 4790 Paderborn

Abstract. After a definition of the term System Level, different system level design activities are identified: product specification and system level modeling, system level simulation, system level analysis and partitioning, embedding into a concurrent engineering environment. These activities then are discussed in more detail with special emphasis on modeling.

1. What is system level?

Today in the area of digital hardware design there is a rather 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. 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 concept of abstraction and by this the system level more visible, a bottom up presentation has been chosen. The lowest level (level 1) usually is 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 doesn't constitute an own level. The **switch level** (level 2) is the next abstract one. This level is rather 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 concpt to model switch level circuits by finite automata. 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. In the ideal case the value domain is restricted to Boolean values 0,1 while

J. Mermet (ed.), VHDL for Simulation, Synthesis and Formal Proofs of Hardware, 259–276.

the time domain remains continuous. Again the problem of uncertain values forces 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 come 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. 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 the counting of clock ticks. So the time domain now has become a discrete one as well. The register transfer level is very helpful in clean synchronous designs, it forces somehow to 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, ME73, ME85], CDL [CH79], DDL [DD79], KARL [HA77]) are used rather in universities only. 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 usually are 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 usually is 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 as in usual algorithmic languages. Algorithmic languages, have been a purely academic area for a long time. The increasing complexity of digital systems and caused by this the need for high level synthesis tools make this level more and more at-

tractive 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] and partially VERILOG [TH91] and VHDL may serve as examples. Finally at the **system level** (level 6) it is abstracted from the algorithmic implementation of system's instruction sets. 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. E.g., in the case of a usual processor th 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. Dependent 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 in a totally distributed manner the different components decide due 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.

Up to now the levels of abstraction have been looked at from the pure electronics point of view. However, very rarely pure electronic systems are 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 usually doesn't play the dominant role but a serving

one (it's the engine's power a car driver is interested in and not the operation of the electronic controller). There may exist levels of abstraction within the modeling of non electronic parts as well but they are of minor interest for our discussion. It can be assumed that at the highest level a separation into the different domains of engineering takes place. This partitioning is a highly complex action, up to now nearly completely carried out by human decisions, based on some expert knowledge. Once the partition has been selected and the interfaces have been defined the further design can be carried out within the specific engineering domains. Now the common interfaces may get an individual interpretation and modeling in the different areas. From now on a point of view centered at the specific areas is legal. From our point of view after the step of par ioning an electronics system is obtained, providing an instruction set for software (which is of minor interest for the further steps of electronic design) 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 priory 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 have been selected the specification of the electronic component to be designed (if not already existing) is obtained.

2. What is 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 to become partitioned it is necessary to have a model of the entire system. Therefore modeling is essential, not only but especially at this level of abstraction. Such a model is the initial reaction to a specification of the entire system. This specification should be independent from the solution selected to find an implementation. The obtained model being the first formal document describing the system to be built some kind of verification is essential. It might be possible to proof formally (a better word is analytically) that certain aspects of the specification is met. The main activity to check the correctness of the initial model with respect to specification is simulation, i.e. experimenting whether the designer's intention is matched. 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. All these design tasks being performed, the already 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) interaction with a concurrent engineering environment.

These activities now are discussed in a somehow more detailed way.

3. System level specification and modeling

3.1. General remarks

There is no "one and only" system level modeling method and by the same reason no "one and only" specification method. Requirements engineering being a complex area by 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 the ideal case the basic characteristics of a system to be build are fixed, only some parameters have to be supplied. Examples for such approaches are systems like DEBYS [BK91] that allow to fill in these parameters in an interactive way that is supported by a knowledge based approach. The result of this search process through a limited (but very large) design space then can 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 requirements engineering will not be possible. It even will not be possible to formulate homogeneous system models. In 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 this one that is most familiar to the designer. In this section therefore simply a couple of modeling approaches shall 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.

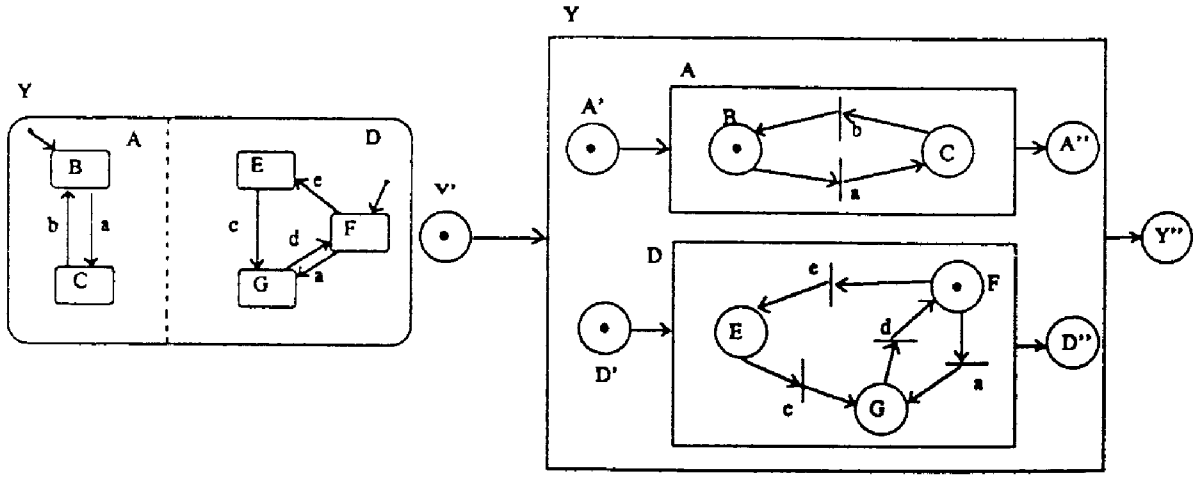

Fig. 1. Example of a statechart and equivalent S-net (from SU90)

3.2. Statecharts and structured Petri nets

Both, Statecharts and Structured Petri Nets (S-Nets), originate from finite state machines (FSM). Both add two major concepts for handling complexity: hierarchy and concurrency. And finally both are graphical approaches for system specification. 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 a FSM as well, and so on. 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 be always the same one (default initial state) or be dependent on the internal state a macrostate has been when this macrostate has been deactivated. The latter situation is supported by the so called **history mechanism** that may be extended recursively to deeper hierarchy levels. The second concept added by Statecharts to FSMs in order to handle complexity is **concurrency**. Several Statecharts now are allowed to operate concurrently. I.e. when the macrostate they are embedded in is activated more than one internal FSM is activated. 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.

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 a 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 life. 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 a 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. It depends on the special situation which concept is the more natural one. The history mechanism is missing in S-nets as originally defined by L. Cherkasova but can be added easily.

3.3. 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 which indicates that it is especially suited for telecom applications. However it is general enough to be used in other areas as well. E.g. [GL 91] discusses the use of SDL in electronic design, relating SDL to VHDL.

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. 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 play a role now is specified precisely. This multiview

approach of SDL make individual descriptions very easy to understand. To get a complete understanding of a complete system, however, it is necessary to combine several subdocuments and to understand the intercorrelation. Like Statecharts SDL supports only asynchronous communication but synchronous communication (rendezvous) can be simulated easily by this concept by explicitly describing a handshake procedure. SDL is in practical use worldwide. Recently extensions towards object orientation (**OSDL**) have been proposed [MB87].

3.4. 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 intented 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. Like SDL in GRAPES the static structure of a system is represented by special diagrams. The equivalent to SDL's BD is the **Communication Diagram (CD)** in GRAPES. In addition the structure of a connection within a **CD** is further explained using an **Interface Table (IT)** that specifies the used channels and data types. The basic specification technique for behaviour is given by **Process Diagrams (PD)** with the same meaning as in SDL. There is no equivalent to SDL's **SC**s. On the other hand more emphasis is laid on the specification of data objects and data types. For this purpose **Data Tables (DT)** and **Data Structure Diagrams (DD)** are used. The counterpart to **IT**s from the process' point of view is the **Specification Diagram (SD)** where the interfaces of procedures and functions and the export interfaces of modules are specified. The entire declaration hierarchy of a system is represented by means of **Hierarchy Diagrams (HD)**.

3.5. 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 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 [CM84] and its derivates become a useful language for specific aspects of system level specification

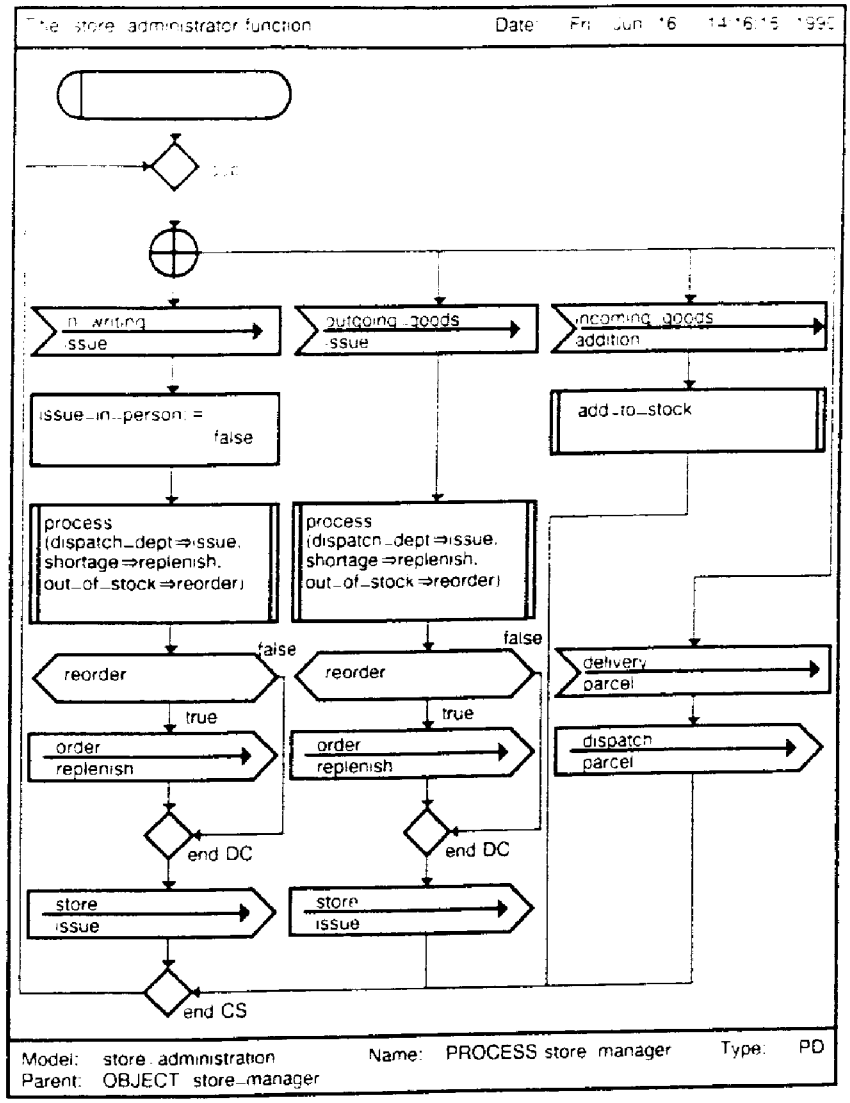

Fig. 2. Example of a GRAPES PD (from HE90)

and modeling. 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 are 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 $H :- G_1, G_2, \ldots, G_n | B_1, B_2, \ldots, B_k$. 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 in 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. Concurrent logic programming has been used for hardware specification [SU85, WS87]. Efficient parallel implementations are available today [GH91].

3.6. Functional programming

System modeling 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 modeling 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 modeling without supporting a functional point of view makes rarely sense. An excellent example for an approach to cover wide areas of electrical systems by functional techniques is GLASS[SE90]. Functional programming not only is 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 rather simply on functional specifications while the same is very complicated in the imperative domain. Therefore it is not surprising that functional programming is playing a more and more 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.

3.7. Object-oriented programming

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** is used that is present in most OO languages. By this principle hierarchies of classes can be built. Rather 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 are 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 understood by the server. Technical systems behave in a completely different way, obviously.

3.8. Computer hardware description languages for system engineering

CHDLs like VHDL have been designed to cover a specific aspect of system 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 already have identified that a multiparadigmatic approach seems to be the most promising one for system modeling, CHDLs get 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 Statecharts, the fine grain functionality of the states, i.e. the operations to be performed in the states, are specified in VHDL. Another approach to add system design capabilities to VHDL is VAL [AG88]. Here additional information is added to allow to reason about provided descriptions. There are other CHDLs that offer more direct support for the system level. DACAPO III [DAC87] 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 [GL90, 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 of the intended polyparadigmatic approach is the concept of CONLAN [PB83] to provide a deduction system for building modeling languages instead of just defining one language. A revival of CONLAN might be a very promising way of approaching a framework for system level modeling.

4. System level simulation

By the heterogeneity of typical system level models, monolithic simulation systems hardly seem to be adequate. Therefore multisimulator-systems may 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 rather 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 happens, this point of time is legal for all simulators involved. There are two extrem 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 happen the event in the closest global future to 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.

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 analyze experiments have to be offered. For this purpose AI techniques have been proposed by a couple of authors [AP86, EO86, ER88, LA86, PF91, SM85]. With such techniques a knowledge based automated experimenter can be implemented that perform experiments in a goal-oriented manner. A very advanced system for this purpose has been designed and implemented by H. Pfaffhausen [PF91]. Such a system is a step to close the gap between simulation and "formal" methods.

5. System level analysis and partitioning

Besides experimenting with a modeled 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 in rarely supported at system level. Rule based approaches like this one described in [BI91] may be promising. Similar approaches may serve for an analysis of fabricatability 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. Therefore feature oriented retrieval operations on libraries of rather complex objects are 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.

6. Interaction with a concurrent engineering environment

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, fabricatability, marketing aspects, substitutability of

preproducts, management cost, recycling costs, legal aspects, etc.

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 provided local optima. A concurrent engineering model therefore has same similarities to 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. 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. 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 retrieve efficiently 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 concurrently are working 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. 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] and commercial products are emerging. One of the most advanced solution for the framework problem is the JESSI Common Framework (JCF) [KK91].

7. Conclusion

Traditionally systems have been designed by partitioning the problem into different engineering domains manually. Within these domains the system's parts then have been designed independently with the methods of different areas. The well known problem of life-cycle reduction, increasing system complexity and market pressure make a more systematic design process at the system level necessary. Such a design process has to be supported by computer based methods. Finally an integrated CE environment has to be achieved. Today the necessary base technology is evolving. So the problem can be attacked now. And it has to be attacked, especially when the european

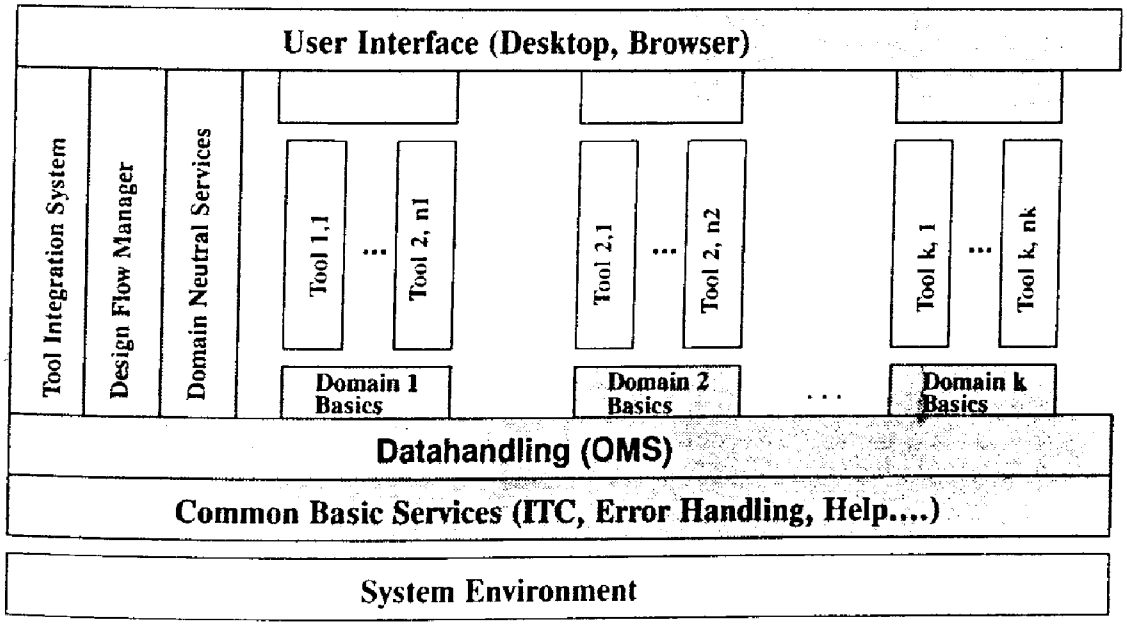

Fig. 3. Architecture of an advanced concurrent engineering environment (JESSI Common Framework)

situation is considered. Europe today is very strong in system integration and the design of highly complex systems of heterogeneous nature. To maintain this strength the understanding and application of automated system level design and concurrent engineering seems to be essential.

8. 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. In: RW91

[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

[CM84] W.F. Clocksin, C.S. Mellish: Programming in Prolog. Springer 1984

[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

[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. In: RW91

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

[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

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

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

[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.): GRAPES Language description. 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

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

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

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

[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, 1970

[ME73] J. Mermet: Etude méthodologique de la conception assistée par ordinateur des systèmes logiques. Thèse d'Etat, Grenoble, 1973

[ME85] J. Mermet: Several steps towards a Circuits Integrated CAD System: CASCADE. North-Holland, CHDL 85, Tokyo, 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

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

[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

[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

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

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

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

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

[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

[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

[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

[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 Disciplin of Computer Program and Design. Prentice Hall, 1979