Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

PROCESS CONTROL FOR LARGE COMPLEX SYSTEMS

Felix Bretschneider, Christa Kopf, Markus Zolg

Corporate Research and Development
Siemens AG
8000 Munich 83

ABSTRACT

The ever-increasing complexity of technical systems, especially computer-based ones, calls for support to control these systems efficiently. This paper deals with the problems of modeling and managing complex processes. A new modeling technique which combines both process and object characteristics is presented and a scheme for automatically applying the model information to process control is described. The approach has been been tested in design systems for electronical CAD.

1 INTRODUCTION

The term "system" is often used in conjunction with computer based technologies. Computers offer the power to control, simulate, or design systems of an unforeseen complexity.

The goal of early computer support was to provide solutions for singular problems which could not be handled manually such as finite element methods or electronic simulation. More and more of these "tools" became available, thus covering entire problem domains (e.g. electronic design). The tool sets began to form intricate "systems" themselves and confronted users with the task of selecting appropriate tools for a given problem and coordinating their employment. This task itself has reached the limit of being manageable "manually" and calls for computer support.

The crucial prerequisite for the step from manual heuristic handling of systems to highly automated solutions lies in the understanding and, subsequently, a formal modeling of these systems. Formal system modeling techniques have been a central subject of systems theory since the early 1960ies (Petri,1962. Hopcroft and Ullman,1967. Zeigler,1976 and many others). A first approach to the required system modeling task is the application of these well-established theories. As we will show in Section 2 of this paper they provide solutions for specific system aspects. However, we lack a comprehensive system model which provides enough detail for the control of complex processes. Especially we miss a modeling technique which integrates process flow and object structure. In Section 3 we examine some of the most popular modeling techniques and present our approach to this problem. Based on this approach Section 4 shows how to deal with the the special aspect of decision making within processes. In Section 5 a process management model is presented.

2 PROBLEM DOMAIN

2.1 Overview

Systems in this context are defined as a set of mechanisms or procedures transforming matter or information.

Traditionally the characterization and modeling of these systems, which is a prerequisite for their control, follow two somehow complementary approaches:

a) The *process flow* is regarded as the central aspect of the system. Process flow refers to the structural and causal order of the steps/phases which the matter/information has to pass.

b) The system is described via the *characteristics (e.g. structure) of the matter/information* as it is processed (and changed) within the system.

In this paper we will restrict ourselves to the observation of *discrete event information systems*, meaning that

i) the process is structured into a finite number of discrete steps (in contrast to continuous phases). Each step is performed by an agent which is activated depending on the occurrence of specific events.

ii) the system processes information which can be described as a collection of discrete items in the following referred to as *objects*.

Examples of such systems are hardware and software design systems where the information objects are representations of the entity to be designed (e.g. source code, object code, executable) and the agents are the tools used to achieve the design goal (e.g. editor, compiler, linker).

The system modeling approach according to a) tries to answer the question *"What task does a specific agent perform, and how do the individual agents work together?"*, whereas approach b) raises the question *"What do the object and, especially, their structure look like in a specific state of the process?"*

In the following paragraphs we will look at object structures and process flows in discrete event information systems more closely and and discuss different modeling techniques and their limitations.

The complexity of the *object structures processed* varies strongly among systems. It depends on the number of object types, their mutual dependencies and the space in which these dependencies occur. A good example for systems with complex object structure are hardware design systems. The design objects (ICs, modules, cells) can be represented in a three dimensional "design space" (Bretschneider, Lagger, Schulz 1989). This space is spanned by the object hierarchy (e.g. an IC is composed of many modules which in turn are made up of cells), by the versions of components (representing different design stages) and alternate solutions (showing different design approaches), and by different representations/views of these components (behavior, structure, physical layout)). Appropriate design data management has to handle the identification and localization of each of the data objects describing specific system aspects. But, even more a task for computer based solutions, the design system has to resolve complex mutual dependencies of data objects in this space.

Complex process flows, besides pure sequences, contain non-trivial control structures such as recursions, iterations, choices, concurrency and synchronization. Thus many dependencies of agent executions are introduced which have to be described and supervised adequately. Again, hardware design systems provide a good example for systems with complex process flows. The different tasks (synthesis, simulation, test pattern generation, floor planning, layout assembly), due to their complexity, are often handled sequentially resulting in many design iterations, in a large number of design versions and, because of inconsistencies, in error-prone designs. This problem is even more severe in systems design where different domains (hardware, software, mechanics) have to be coordinated. Thus "concurrent" design has become a hot topic which allows to proceed with many design steps in a parallel and which provides detailed coordination among them.

Figure 1 shows a classification of systems with respect to complexity of process flows and object structures. It also shows typical examples. As can be seen from above, the systems we are most interested in, CAD systems, exhibit *both* complex flows *and* complex object structures. In order to control such processes we therefore need an integrated modeling technique which adequately describes both aspects. But let us first look at some conventional, well established modeling techniques and their limitations. Commonly used modeling and process control tech-

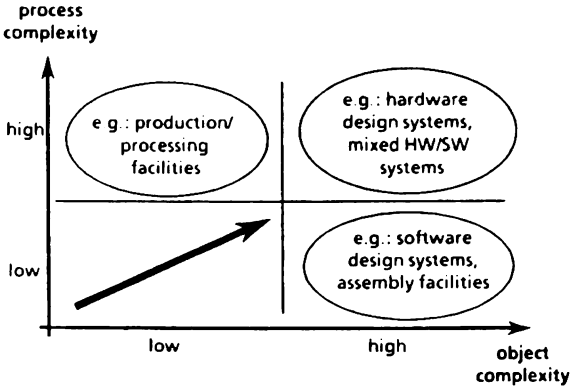

Figure 1: System And Process Characteristics

niques are *either* object structure *or* process oriented. A typical example for the first category is given by the UNIX^{TM} *make* facility commonly used for software design (UNIX^{TM} Manual Pages(1)). A typical *make* structure for compiling and linking a program is shown in Figure 2 a). The key items in the *make* scripts are the module names, the script structure represents the program structure, and the commands - often representing one and the same process step e.g. "compile" - are bound to these structural items. Thus, due to the structure orientation, the model of process flow is entirely broken up. Without knowing the inference strategy of *make* it is not possible to find out which step is performed first and which one later, or even whether there is a potential deadlock in the system. Another severe limitation is that a new *make* script is needed for every new program structure. *make* is good for (the final stages of) software design because in this case the object structure is known and remains fixed. Complementary approaches like flow graphs or state-transition networks model the "process flow" but neglect the object structure entirely. A flow graph models a process as a directed graph

make-stucture:

<module>: <depends-on-module(s)>

action (if module older than depends-on-module)

e.g.:

program: $\textbf{object}_1\ \textbf{object}_2 \bullet\bullet\bullet \textbf{object}_n$
'link'

$object_1$: $\textbf{source}_{11}\ \textbf{source}_{12} \bullet\bullet\bullet \textbf{source}_{1n}$
'compile'

$object_2$: $\textbf{source}_{21}\ \textbf{source}_{22} \bullet\bullet\bullet \textbf{source}_{2n}$
'compile'

a)

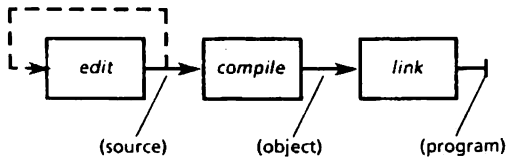

Figure 2: a) Structure and b) Process Oriented Model for Software Development

of items which represent individual activities. Thus the model provides information about the sequential respectively concurrent order of activities, also indicating which activity feeds its "output" as "input" to a following activity. There is no information modeled concerning the structure of this input and output. A simplified software development flow is shown in Figure 2 b) as a complementary view to the structure oriented *make* model. Flow graphs are typically used to describe the functionality of systems to human beings (e.g. program flow charts) or to handle very simple objects (e.g. description of the evaluation of a formula). As can be seen from the software design example in Figure 2 b) they cannot directly be used for control.

State-transition networks such as standard Petri nets (Rozenberg, Thiagarajan 1986) or finite state machines (Hopcroft, Ullman 1988) also allow for an appropriate control of asynchronous respectively synchronous processes but do not support the structural modeling of the processed information either. These models are discussed in more detail in Chapters 3.1.1 and 3.1.3.

2.2 Integration of structure and process

The ambitious goal of our project is a synthesis of both approaches described above, i.e. a combination of process flow and object structure modeling. In the following paragraphs we will illustrate why we regard this integration of process and object structure as essential and we will show why the respective approaches cannot simply be put together in an orthogonal way.

An orthogonal process-object structure space allows for navigation in this two dimensional space, which is indifferent to the choice of a preferred axis. In Figure 3 the definition of a process flow, which is applied on the entire object, has the same result regarding a specific object component (trace a) as an initial selection of the latter component and a subsequent application of the given process flow (trace b) Unfortunately, general complex systems do not nec-

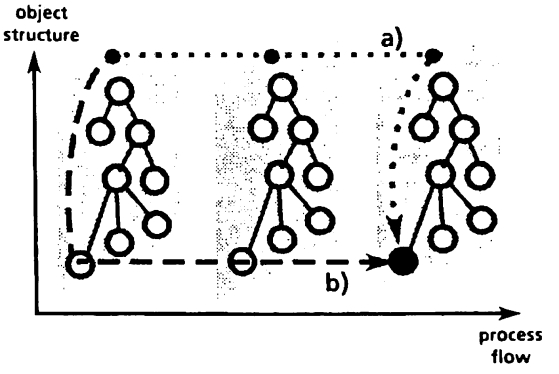

Figure 3: Orthogonal Process - Object Space

essarily comply with this simple orthogonal model, as we will see in the following discussion.

Firstly let us examine the underlying principle of an approach according to trace a) of Figure 3. A process flow oriented model will have no knowledge about the structure of the system to be modelled. This implies that a single flow is applied to every object component in the same way.
If we take for example the development of a heterogeneous system, say an electronic telephone switchboard which consists of hardware and software subsystems, it is obvious that the individual development of either component will follow entirely different processes.
Consequently, it can be stated that a system model based solely on the process flow is not sufficient, rather the object structure has to be taken into account.

Secondly, trace b) supposes a static object structure. This means the object structure is not changed during the process, the graphs representing the individual object structures at the different process steps are isomorphic. Again, this is an unbearable limitation for process models. Assume for example the physical layout of a circuit might require a partition-

ing into several chips which do not match the boundaries of the circuit modules. In this case the structures are not only non-isomorphic, but it is also not possible to map a limited set of circuit modules to a physical partition, thus yielding a real n-to-m relation between components of either representation.
Concluding, we find that is insufficient, to have a static object structure model, rather it has to be observed how the object structure is transformed during the process.

Both views of systems show the need for an integrated modeling of process flow and object structure, which is more than a combination of isolated approaches dealing with either aspect. Following from the discussion above, such an integrated system modeling is a non-trivial but essential prerequisite for a general approach to the control of complex systems. Our approach to this problem will be discussed in Chapter 3.3. Furthermore, as we will see in Chapter 4, another important aspect of systems modeling addresses the dynamic characteristics of the system structure. Thus, with respect to the integration of process and object structure on one hand and to the degree of flexibility on the other hand our "area of interest" can be located in a graph according to Figure 4. First approaches to the integration of structure and process control have been presented in the CAD-Framework area. E.g. the integration of object and (process-) resource management was proposed (Smith, Cavalli 1990). Other concepts have been presented in the context of concurrent engineering, trying to parallelize certain activities, which requires knowledge of the process and its objects (Pennell, Winner, 1989). Nevertheless it can be stated that the addressed problems are far from being solved, while at the same time modern engineering methods (covering all phases from planning through development to fabrication) require appropriate support exactly in this area.

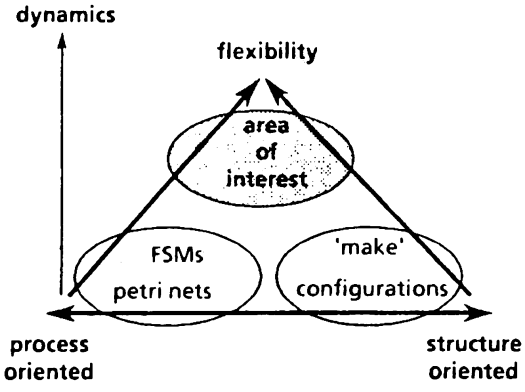

Figure 4: Synthesis Of Structure And Process Oriented Models

3 SYSTEM MODELING

3.1 Modeling techniques

As described in the overview there are several different ways of modeling complex systems. In this paper we discuss three different techniques. To show the differences between them, we model the same example using each technique. The example shows a part of a manufacturing process: Two elements "A" and "B" have to be constructed separately and then have to be assembled. The elements "A" and "B" are independent. The only restriction is, that the construction of each element requires the same tool, which is available only once.

3.1.1 Finite Automata

The mathematical theory of automata is mainly an abstract study, but many applications occur in daily life. A finite automaton is defined as a set of states, a (nonempty, finite) input alphabet and a transition function. A state describes the momentary status of the system with respect to the entire process history. The input alphabet defines a set of possible input symbols or actions. Depending on such an input symbol the transition function describes state transitions (Hopcroft, Ullman 1988).

A directed graph can be associated with a finite automaton. The nodes of the graph correspond to the different states. One state is supposed to be the starting state, some other states are marked as possible final states. The state transitions are represented as directed arcs, leading from one state to another, labeled with the corresponding input symbol.

The finite automaton accepts its input, if there is a way through the corresponding graph (leading from the initial starting state to a final state) in the sense that the input symbols match with the corresponding state transitions. Figure 5 shows the example mentioned above modeled as a finite automaton. Looking at the graph it is not obvious any more, why there is

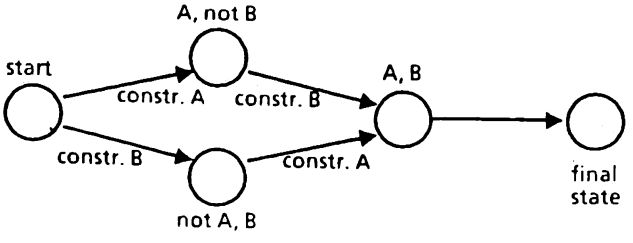

Figure 5: Finite Automata

a conflicting situation after starting the system. One can see only, that the state transitions "construct-A" and "construct-B" have to be executed one after the other (but not why!).

3.1.2 Flow Graphs

Data flow models describe systems in terms of locally controlled events. Each event is the "firing" of an "actor" (Dennis 1985). The semantic of a data flow graph is closely related to the semantics of applicative programming languages. A program module is represented by a directed graph. The nodes of the graph are called "actors", and the arcs are called "links". The links represent paths over which data values are conveyed from one actor to another. The links that terminate on an actor make up an set of input links of that actor. The links that originate from an actor make up an set of output links. The state of a computation in progress is shown by placing tokens on the links of a data flow graph. A computation can be regarded as a succession of snapshots between which tokens are placed and removed by firing of actors according to the following rules:

- An actor of a data flow graph is enabled iff there is a token on each of its input links.
- Any enabled actor may be fired to define the "next state" of the computation.
- An actor is fired by removing a token from each of its input links and placing a token on each of its output links.

Each token has an associated data value, and the firing of an actor includes applying a rule characteristic of the actor to define the values of tokens places on output links in terms of the values of the tokens originally on the input links as shown in Figure 6. Using

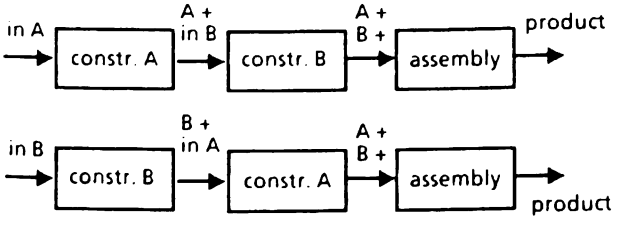

Figure 6: Flow Graph

a simple flow graph there is no way to show the conflict. The process has to be modeled twice in order to reflect the different sequences.

3.1.3 Petri Nets

Introduced by Carl Adam Petri, Petri nets are nowadays widely used in computer science as well as in engineering to model concurrent discrete event systems and to study their behavior (Rozenberg, Thiagarajan 1986). Petri nets are directed bipartite graphs which consist of "places" to represent states or locations, and "transitions" to represent actions. Thus, Petri nets are a combination of flow graphs and finite automata. They represent the actions of a system as well as the states. Tokens can be placed on a state to model that a certain condition is satisfied, an object is at a certain location, etc. The set of all places together with their tokens is called a marking. A system is modeled by a Petri net and an initial marking. Its dynamic behavior is defined by the following firing rules:

- A transition T can be fired (is enabled) iff all input-places (places with arcs leading to T) hold one token and all output-places (places with arcs leading from T) hold no token.
- After the transition has been fired, one token is removed from each input-place and one token is placed on each output-place.

Thus the static as well as the dynamic behavior of the system can be modeled as shown in Figure 7. Moreover, a rich mathematical theory exists for Petri-Nets which allows for proving of certain dynamic properties of such systems, e.g., absence of deadlocks, etc. (Genrich 1987). As shown in our example, the Petri

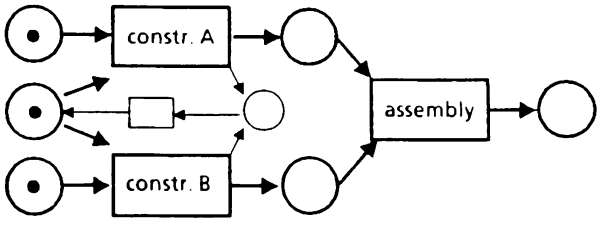

Figure 7: Petri Net

net offers the possibility to express the conflicting situation without having to specify an explicit sequence of events. That means, either "construct-A" or "construct-B" can consume the token representing the construction tool. This enables a more flexible way of modeling.

Comparing the three different approaches of modeling the small example, it becomes obvious that finite automata and flow graphs provide less modeling power than Petri nets. They can in fact be viewed as restricted forms of C/E Petri nets.

3.2 Modeling Requirements

An appropriate technique for modeling complex large systems has to fulfill the following requirements. The model used to describe such a system has to contain descriptions of

1. active components (e.g. agents)
2. passive components (e.g. storage facilities, states)
3. complex objects (e.g. files, current state)
4. static/dynamic dependencies between components and objects (e.g. I/O behavior of a agent)

The behavior of a system is governed by the actions that might occur. An action, i.e. execution (firing) of an active component, usually causes a change in the state of the system. Such an execution can be an agent invocation, a user request, etc.

The model has to reflect sequences, concurrencies and conflicts of actions. In order to be able to control a process, i.e. a number of actions with a given goal, the system model must accurately describe all the dependencies between the actions.

Knowledge about the causal order is important to invoke the right agents at the appropriate time. Knowing about concurrencies makes it possible to start processes in parallel (concurrent engineering to reduce time and costs). A conflict in the system is a situation where a decision has to be made which one of several enabled agents should fire. This decision can be made by a user or a process control facility (see chapter 4).

A modeling technique should provide different levels of detail for describing a system. Using model hierarchies causes a reduction of complexity through data hiding, making a system model easier to understand, and the possible reuse of substructures (just like the introduction of procedures usually makes a program easier to understand and reduces code size). Thus the top level description of a system would just give the information about the main components and their relations without looking at the detailed structure of each component (e.g. subtasks performed by an action).

The goal of modeling a software system is to control the process flow during execution, making decisions on the fly instead of a priori (see also chapter 4). Early decisions unnecessarily limit the search space for solutions often resulting in time consuming and costly backtracking steps later on. "Lazy" decision making postpones all decisions until the latest possible time when hopefully more data is available to make them in a more precise way. If providing an intelligent user support, it is necessary to exactly identify the state of the system in order to be able to make the "right" decisions.

Therefore the model has to express the dynamic behavior of the system by describing the state changes caused by the firing of transitions. For example states of the systems could be described with the help of tokens and the state changes could be expressed by firing rules causing some tokens to be removed and others to be created.

In addition to the requirements above, a system model must be able to represent detailed information regarding the objects that are manipulated by the system. Their structure and relationships often influence the process flow. Consider design systems, for example, were the hierarchical structure of the design objects has a severe impact on the design process. Some steps are performed top-down were the process on a lower hierarchy level is dependent on the existence of certain object representations on the higher level (typically planning steps), others follow a bottom-up approach (analysis steps, e.g. simulation). Due to this fact, it is necessary to use a modeling technique which allows the identification of individual object types.

3.3 Our Modeling Technique

While modeling a complex system we have to take care of the process flow (the actions) within the system as well as regarding the states of the system. Since Petri nets represent both, the actions and the states of a system, we decided to chose Petri nets as a modeling technique.

Using the simplest Petri net model, the *Condition-Event-Nets (CE-nets)*, we would have the restriction, that each place can contain at most one token at any time (since it models a condition that either holds or does not hold). Describing systems using CE-nets would cause very large models, and the type of systems that can be modeled would be restricted.

A more suitable Petri net type is the *Place-Transition-Net (PT-net)*. PT-nets offer the possibility to use any number of tokens on one place but the tokens are all of the same type. It is therefore possible to model structures like infinite counters.

Since we want to model complex systems processing complex objects, we need the possibility to model essential aspects such as relationships between tokens as described in section 3.2. That means we have to identify the token in order to reflect the structure of the objects. Therefore we use another extension of Petri nets, the "high-level-nets". Within high-

level-nets it is possible to identify the tokens. Thus relations between tokens can be expressed, such as hierarchical dependencies between data objects (e.g. the cell-subcell relationship in electrical CAD). Many variations of standard Petri nets have been studied to provide more modeling power.

We chose *Predicate-Transition Petri-Nets (Pr/Tr-nets)* (Genrich 1987) which support arbitrary types of tokens. This makes it necessary to label the arcs with patterns, identifying the types of tokens that can be used or that are created when firing a transition. In these labels variables can be used. Variables that occur in labels of arcs leading to and from a transition have to be instantiated with the same value during the firing of that transition. An example is given in Figure 8. In the left picture, transition E can be fired since for $x = a$ and $y = c$ the input-tokens satisfy the pattern [x, y]. The right picture shows the situation after the firing. The tokens of the Pr/Tr-net have to

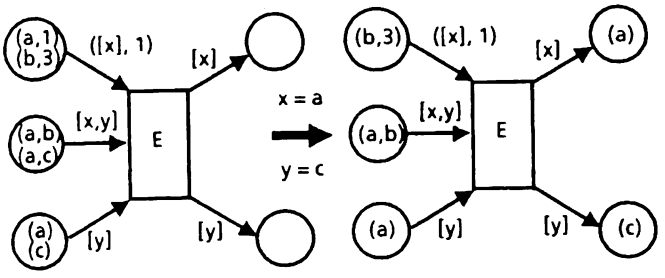

Figure 8: Example of Transition Firing in a Pr/Tr-net

reflect the information concerning the current object in process. That means, a token is a representation of a data object, holding the structural information of the appropriate object. Therefore, the labels characterizing a token contain much information. The labels are tuples of expressions. Each expression represents one component of the actual structure (e.g. type-information, number of parameters, etc).

However, there are still certain unsolved problems concerning modeling large systems with Petri nets. The execution of a transition in a Petri net is data driven, that means an agent can be started, if the required tokens are available. Basic operations like a logical 'not' and 'for all' loops are very often required to describe a complex process but these cannot be modeled directly within Pr/Tr-nets. Additional transitions and tokens can be introduced though to circumvent these problems. Thus the 'not'-problem can be solved by using a token representing the "absence" of objects (rather than its "presents") and 'for all' loops can be simulated by special counters. Nevertheless these additional constructs make the modeling task very tedious.

Another problem is the representation of agents with varying input/output types. These agents have to be modeled several times (since only fixed I/O behavior can be represented in Pr/Tr-nets)

Nevertheless our experience shows, that Pr/Tr-nets are a convenient tool for system engineers and users to define and modify the interaction among agents in a complex system. They also provide a good reference for novice users to learn about system structure and its processes. Furthermore, Pr/Tr-nets can easily support the modeling of concurrent agent executions and conflicts.

4 MODELING DYNAMIC DECISIONS

So far the problems of modeling complex processes and objects were discussed and a first solution for an integrated modeling technique based on Pr/Tr-nets was presented. Such a net model, however, usually contains many conflicting and concurrent transitions. Intelligent process management is expected to support selection and scheduling of individual objects and agents as well as of entire subprocesses *dynamically at runtime*. Therefore the knowledge regarding these decisions has to be adequately represented in the process model. Further knowledge regarding the parameterization of tasks, planning and supervision of constraints has to be added as well.

Different knowledge representation schemes are possible. We chose rules, since they are very flexible and do not require an explicit control specification (*when to do what ?)*, which is seldom evident in decision making situations.

Consider for example the problem of configuring a computer system. One has to select a central processing unit, a bus architecture, memory and I/O units to meet given requirements (performance, size, cost). It is relatively easy to specify the various constraints in the form of rules, such as

$$\begin{gathered}\text{higher performance required} \Rightarrow \\ \text{increase } \mathbf{clock\ rate} \vee \text{choose faster processor} \\ \text{memory access time} > \mathbf{clock\ rate}^{-1} \Rightarrow \\ \text{select memory(access time} \leq \mathbf{clock\ rate}^{-1})\end{gathered}$$

Note that these rules contain linked parameters e.g.**clock rate**. Finding an efficient algorithm (a sequence of actions) which selects an architecture satisfying the given constraints can be very complex because one has to specify the order in which the individual constraints are evaluated. A rule-based system eliminates this requirement.

A further advantage of rules is the possibility to integrate knowledge from different sources relatively easily. This is important for our application since the knowledge regarding process control usually comes

from many different origins (supplier of agents, system integrators, system users, learning mechanisms).

Finally, by using rules, it is possible to implement a simple knowledge maintenance scheme by monitoring the success and failure of each rule application with respect to achieving a given goal. The "success rate" can be used to select the most "promising" rules and to discard "unsafe" ones, thus introducing a very limited "learning ability" to the system.

Rules add a new dimension of flexibility to process modeling and control. But on the other hand they have to be handled with care since big rule sets get easily out of control when it comes to keeping them consistent. Because it is very difficult to find out which rule is applied *in a specific situation*, it is very difficult to "debug" rule sets. Therefore major efforts have to be spent to keep rule sets as small as possible and to introduce some structure on them. We will discuss the modeling of local and global scoped rule sets and its impact on this aspect below.

Traditional rule-based languages, like OPS '83 or Prolog, have only limited expressive capability which sometime makes the task of formulating complex rules very tedious. Moreover these languages are too abstract to be used by system engineers and system users directly. Therefore, on the long run, we will have to look for higher level representations tuned to our application of controlling processes.

Given the choice of a modeling technique, the question arises, how to combine the decision making rules with the net representation of the process:

One solution is to define *local decision making agents*, which are represented as transitions of the net and which produce the respective tokens depending on the outcome of the decision. This approach guarantees that the knowledge bound to a transition is only used when the respective transition fires. Thus, debugging rule sets is made easier. If the user (or a learning system) finds, that a decision was not correct, he or she (it) knows exactly were to look for incorrect knowledge. There is a problem of keeping decisions independent since they must not interfere with the deadlock-freeness of the net. An example of dependent decisions is given in Figure 9. There the places P_1 and P_2 model limited resources which create conflicts because the transition T_1, T_2 and T_3 compete for the only token available on each of these places. If the decisions at P_1 and P_2 were made independently then T_1 and T_2 could both be selected leading to a deadlock since only one of the transitions can actually fire. We see that the possibility to model local decisions depends on the topology of the net in that area. A solution to avoid such problems by restricting the net topology has been described in (Pagnoni, 1990).

Furthermore, planning and constraint supervision in

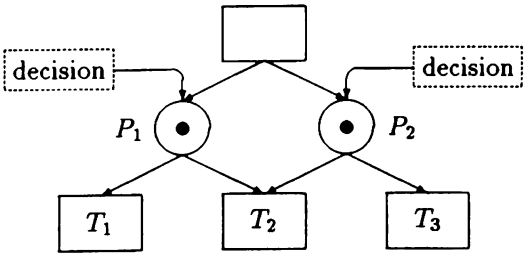

Figure 9: Dependent Decisions Lead to Problems

the local decision making approach is difficult, since the required knowledge is distributed over the net.

This leads to an alternative approach of a *centralized* decision making and planning agent which operates on its own global knowledge base, independent of the net. In this case planning and global control is easier because all process knowledge is uniformly represented and globally accessible. It is just as easy to formulate a constraint about certain properties of the objects that are handled in the system as it is to specify a rule about the process history. However, keeping the consistency between the net and the knowledge base is a real problem. Moreover, because of the size of the knowledge base, debugging is by far more difficult than in the first approach.

Our *"meet in the middle" strategy* combines the two approaches. The knowledge is kept local to the nodes of the net (user friendly) but is compiled together with the net information into a uniform knowledge base (avoids problems of "distributed" knowledge). Prior to this compilation several consistency checks are performed. Reasoning based on the local input information of a transition (e.g. a certain data object) as well as on global data (e.g. the process history) is possible. Figure 10 gives an overview of the architecture of our system, which has been integrated and tested in an existing design system for electronical CAD called HILDA [see (Bretschneider, Kopf et. al. 1990)] .

A graphical process model in Pr/Tr-net representation, annotated with additional *node bound* rules is entered into a graphical editor and simultaneously checked by a syntax checker. An example for such a model is given in Figure 11. A selection between two producers for one object is modeled there. The rules for making the context specific decision depending on producer characteristics (cost vs time) is shown.

After completion of the editing process the model passes through a semantic checker and on positive outcome is finally translated by our Petri-Net compiler into an OPS'83 rule set. For process control the OPS inference engine operates on the rule set (and its

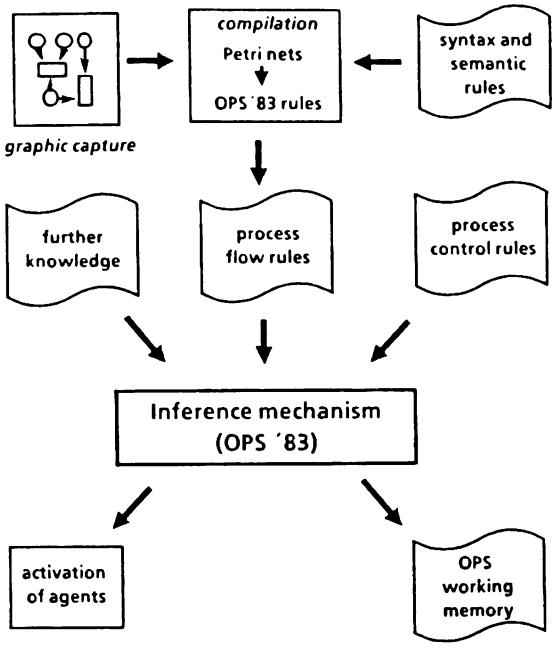

Figure 10: Our System Architecture

own internal working memory) thus activating the individual agents at the appropriate time.

5 PROCESS CONTROL

As stated above, process control in our system is implemented by an inference engine operating on the rule set derived from the process model. Additional rules which control the activation of agents, process their return values, keep the process history and supply means for saving / restoring of tokens have been added.

For the representation and visualization of the process execution we use an extended Petri net trace, which shows each object (token) as a circle and each activation of an agent (enabling of a transition, i.e all input tokens are available) as a box. Directed edges between these symbols indicate that certain tokens were used or created by a transition firing. A trace is always acyclic since each object is uniquely created by one agent. Figure 12 shows a net and two possible traces. One can easily see that there is a fundamental difference between the two representations. The net view gives the static view of the system (i.e. dependencies between agents and data), while the trace represents the execution of one process within this system. For a design system, for example, the net represents mainly the system administrator's point of view (the availability of tools and their integration), while the trace shows the state of a current project to the designer.

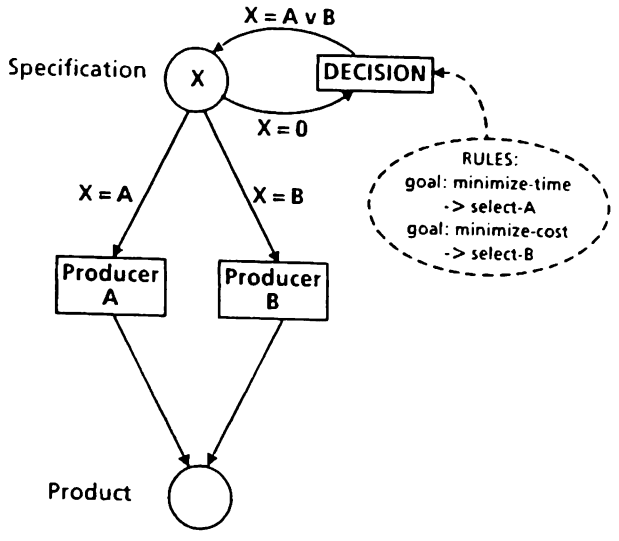

Figure 11: A Simple Process Model

Compared to the representation used by (Casotto 1990) our traces show all activated transitions (all transitions which could have been fired, not only those which were actually fired) and thus keeps a history of all selections including the alternate ones that were not chosen. With the save and restore features for tokens and the supplied replay mechanism, a user can always go back to a previous situation and use an other transition, if desired. Fig. 12 shows such a situation. In the first trace one can see that there has been a choice between transitions A and B and that B has been selected (the "unused" possibility A is indicated by dashed lines). In the second trace a "roll-back" to the first state has occurred and the other alternative A has been explored. Since the information of all created tokens and transition firing is kept it is possible to go back to alternative B later on. The same approach is used for the decision making phases. All instantiations of rules are preserved for inspection by the user or for learning mechanisms.

One can see that traces provide a nice graphical view of process executions. For complex processes (such as CAD) one would not look at the whole trace at a time, but use different views (hierarchy, flows, state diagrams).

6 SUMMARY AND OUTLOOK

We have described the problems of modeling and controlling complex processes and presented a new approach using Predicate-Transition nets and rules. While this method solves some of the problems involved, many questions remain. These can be summarized as follows:

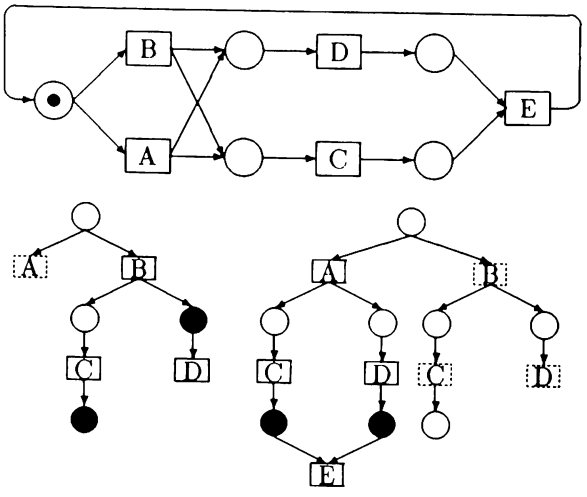

Figure 12: A Petri Net and its Traces

- increased modeling power (synthesis of object structure and process flows, representation of "embedded" decision knowledge)
- advanced process control (high-level decision support, planning)
- application of learning methods (knowledge acquisition and maintenance)

Besides continuously testing our methodology in different target applications our future research will concentrate on decision support and planning since there is a strong need to support the users of complex systems, e.g. CAD systems, in this respect.

REFERENCES

Bretschneider, H. Lagger, and B. Schulz. 1989. Infrastructure for Complex Systems - CAD Frameworks. In *Computer Aided Systems Theory - EUROCAST '89*, eds. F. Pichler et.al., 125-133. Lecture Notes in Computer Science **410**, Berlin: Springer-Verlag.

Bretschneider, F., C. Kopf, H. Lagger, A. Hsu, E. Wei. 1990. Knowledge Based Design Flow Management. In *ICCAD-90: Digest of Technical Papers*, 350-353. Washington: IEEE Computer Science Press.

Casotto, A. 1991. *Automated Design Management Using Traces.* PHD-Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Dennis, J.B. 1985. Data Flow Computation. *Control Flow and Data Flow: Concepts of Distributed Programming, Proc. of the NATO Advanced Study Institute*: 345-354.

Genrich, G. 1987. Predicate / Transition Nets. In *Petri-Nets: Central Models and their Properties*, eds. W. Brauer et.al., 207-247, Lecture Notes in Computer Science **254**. Berlin: Springer-Verlag.

Hopcroft, J.E., and J.D. Ullman. 1967. An Approach to a Unified Theory of Automata. *Bell systems Technical Journal* **46**, 1793-1829

Hopcroft, J.E., and J.D. Ullman. 1979. *Introduction to automata theory, languages and computation.* Reading, MA: Addison-Wesley

Pagnoni, A. 1990. *Project Engineering.* Berlin: Springer-Verlag.

Pennell, J.P., Winner, R.I. 1989. Concurrent Engineering: Practices and Prospects, *GLOBECOM '89, IEEE Global Telecommunications Conference and Exhibition.* **vol. 1**: 647-655

Petri, C.A. 1962. *Kommunikation mit Automaten.* Schriften des Institutes für instrumentelle Mathematik, Bonn.

Rozenberg G., and P.S. Thiagarajan. 1986. Petri Nets: Basic Notations, Structure and Behavior. In *Petri Nets*, 585-668. Lecture Notes in Computer Science **224**, Berlin: Springer-Verlag.

Smith, R., Cavalli, A. 1990. Building a Fourth Generation Framework. it High Performance Systems **vol.11 no. 6**: 63-67

UNIX^{TM} Manual Pages(1)

Zeigler, B.B. 1976. *Theory of Modeling and Simulation.* John Wiley and Sons

AUTHOR BIOGRAPHIES

FELIX BRETSCHNEIDER received a masters degree in computer science from the Technical University of Munich in 1987 and joined Siemens Corporate Research in the same year. He has been working in the field of computer aided design systems for four years. His current research fields are engineering management and design decision support.

CHRISTA KOPF received a masters degree in computer science from the University of Kaiserslautern in 1990. Starting with her master thesis she has been investigating net-based modeling techniques for design processes. She now works in the design automation group of Siemens Corporate Research.

MARKUS ZOLG earned a masters degree in Electrical Engineering from the Technical University Aachen in 1987. Subsequently he joined Siemens Corporate Research, working on system architectures. During the last two years his research fields have been engineering management and infrastructures of computer aided design automation systems.