A Cognitive Model of Design Processes for Modelling Distributed Systems

Olaf Schröder
Institute OFFIS, Escherweg 2, 26121 Oldenburg, Germany
Claus Möbus
Department of Informatics, C.v.O. University, 26111 Oldenburg, Germany
Knut Pitschke
Department of Informatics, C.v.O. University, 26111 Oldenburg, Germany

E-Mail: {schroeder, moebus, pitschke}@informatik.uni-oldenburg.de

Abstract. Condition-event Petri nets are a means to model technical, social, and natural processes and organizations in order to unterstand their behavior, to identify bottlenecks and resource shortcomings, and to propose appropriate changes. PETRI-HELP is an intelligent problem solving environment that supports this modelling activity. Since there is no clear-cut domain theory of Petri net modelling, net design processes can be supported only in limited ways. In order to overcome this situation, a cognitive model of problem solving, knowledge acquisition, and knowledge modification was developed which is an instance of a general theoretical framework. The main results of the model can be summarized in three hypotheses: Problem solving and knowledge acquisition in a domain without a worked-out domain theory
1. consists of i) applying weak heuristics and acquiring new knowledge in response to impasses, and ii) knowledge optimization
2. involves few simple, fairly general heuristics which seem to be important also to other design and configuration domains
3. can be supported by giving feedback and help information sensitive and adaptive to the actual needs of the learner, if appropriate behavioral indicators are provided.

1. Introduction

Petri nets are a powerful formalism for modelling time-discrete distributed systems. Many natural, social and technical systems can be conveniently modelled by them, and the dynamic behavior of these systems can be studied by simulation: for example, technical devices (like machines, mechanical or electrical devices, electric or hydraulic circuits), production lines, office networks, organizational processes in a factory, in administration, etc. With Petri nets, systems of this kind can be described and analyzed in order to identify faults, bottlenecks, or resource shortcomings. Petri nets are an especially convenient, easy to understand formalism because they visually represent concurrency and synchronization. Their algebraic representation has a clear semantics. So they are also used as semantics of abstract formalisms like process terms (Olderog, 1989), hardware description languages (Damm et al., 1990), and nonsequential programming languages.
For a novice, working with Petri nets can cause several difficulties. Features like the distributedness of processes and the synchronization of concurrent subprocesses are sometimes hard to understand. So design problems in the net domain are rather different from "classical" ITS-problems, like arithmetic, geometry, or functional programming, which for example is supported by our intelligent problem solving environment ABSYNT (Möbus et.al., 1994; 1995). But there are similarities to the domain of parallel programming. Another source of problems for novices is the fact that there are not many approaches to support Petri net design in a systematical way. Even in introductory textbooks (e.g. Reisig, 1985; 1992), Petri nets tend to be presented as ready-made solutions to modelling tasks described informally, but there is no clear cut methodology for their construction. Furthermore, we think that students need to have an opportunity to practice design and problem solving in the net domain. So a computer-based intelligent problem solving environment (Möbus, 1995) should be valuable.
We developed such an environment, PETRI-HELP (Möbus et al., 1992; Pitschke, 1994; Schröder et al., 1993), that enables the learner to create condition-event Petri nets to given specifications of distributed systems, to state

and test hypotheses about the correctness of the given solution proposal with respect to (parts of) the specification, to receive feedback and to ask for completion and correction proposals. PETRI-HELP was created using guidelines which were derived from our ISP-DL Theory (*i*mpasse - *s*uccess - *p*roblem solving - *d*riven *l*earning, e.g., Möbus et al., 1992; 1994) which is a theoretical framework of problem solving, knowledge acquisition, and knowledge modification. It attempts to integrate impasse-driven learning (Laird et al., 1986; Newell, 1990; van Lehn, 1991), success-driven learning (e.g. Anderson, 1989), and phases of problem solving (Gollwitzer, 1990). Briefly, it states that the problem solving process may consist of four phases: The problem solver *deliberates* with the result of choosing or creating a goal to pursue, then a *plan* to reach the goal is created, the plan is *executed*, and the result obtained is *evaluated*. *Impasses* might result because the problem solver is not able to choose a goal, or a plan cannot be created, or execution is not possible, or the result is not satisfying. The problem solver reacts to an impasse by applying weak *heuristics* like asking for help. As a result, the problem solver may overcome the impasse and acquire new knowledge (impasse-driven learning). Knowledge applied successfully is optimized (success-driven learning) so it can be used more efficiently in future.
The ISP-DL approach implies several design principles for an intelligent problem solving environment (see e.g. Möbus et al., 1992): Firstly, free, unconstrained problem solving should be enabled. The learner should not be interrupted because he or she will actively look for help as a weak heuristic when caught at an impasse. Secondly, the learner should be encouraged to make use of her/his own solution ideas. So the system should be able to deal with a large solution space and to tailor the information provided to the actual problem solving phase and knowledge state. Thirdly, help information should be directed to the actual problem solving phase. PETRI-HELP is designed according to these criteria. It *offers* help but does not interrupt the learner. The learner may state hypotheses about the correctness of her/his solution proposals, and PETRI-HELP gives feedback and, on further request, completion and correction proposals and explanations. PETRI-HELP supports even unusual solution ideas by being able to analyze and comment on *any* solution proposal of the learner. This is possible because Petri net solution proposals are analyzed with respect to temporal logic task specifications by model checking (see below). Concerning the problem solving phases described above, PETRI-HELP supports the following sub-activities of Petri net modelling: i) to develop specifications of systems or processes to be modelled ("*deliberating*"), ii) to plan a Petri net solution for a specification ("*planning*"), iii) to actually construct a Petri net ("*executing*"), and iv) to check whether the resulting net meets the specification ("*evaluating*").
In the "classical" ITS domains, there is a domain theory where help and explanations are usually based on. But as mentioned, for Petri nets such a design theory is lacking. Therefore, the completion and correction proposals delivered by PETRI-HELP are based on "ad hoc" rules learned by the system from the actions of its users. These rules are not based on a theoretical and empirical account of Petri net design. A consequence of this is that PETRI-HELP proposes what to add and to delete from the actual Petri net proposal, but does not give enough support to the construction *process*. A related problem is that a general understanding of learners' cognitive processes in constructing models of distributed systems (i.e., Petri nets) is missing. Therefore, the design processes of subjects were investigated empirically by single-subject protocol studies, and based on them a cognitive model of Petri net design was developed which is described in this paper. The model intends to put forward an empirically based design theory. More specifically, it is aimed at the following research questions:

- In a domain without a worked-out domain theory, is it feasible to model problem solving and knowledge acquisition processes within the ISP-DL framework, i.e., consisting of i) applying weak heuristics and acquiring new knowledge in response to impasses, and ii) the optimization of knowledge already acquired?
- In a domain without a worked-out domain theory, is it feasible to model problem solving and knowledge acquisition processes by simple and general, "weak" heuristics requiring not much domain knowledge?
- In a domain without a worked-out domain theory, is it possible to *dynamically* diagnose online the domain knowledge, heuristic knowledge, and knowledge acquisition processes of the learner?
- In a domain without a worked-out domain theory, how is feedback and help information to be designed that is sensitive and adaptive to the actual problem solving and knowledge acquisition processes of the learner?

Answers to these questions should provide i) hypotheses about problem solving, heuristics, and knowledge acquisition processes of Petri net modellers and, more generally, hypotheses about processes in design of models of distributed systems, ii) (parts of) a domain theory of Petri net design that can be used for improved feedback and help information, and iii) a basis for adapting the information offered by PETRI-HELP to the actual knowledge of the user. In the next section we will sketch PETRI-HELP. Then we will describe the model. Most of it is implemented but not yet integrated into PETRI-HELP. We will end the paper with some conclusions.

2. The problem solving environment PETRI-HELP

PETRI-HELP (Möbus et al., 1992; Pitschke, 1994; Schröder et al., 1993) is designed to support novices modelling time-discrete and distributed systems with condition-event Petri nets. PETRI-HELP contains a

sequence of twelve modelling tasks partially ordered according to several modelling goals. The learner may create Petri net solutions to these tasks. Each task is specified as a set of temporal logic formulas (Kröger, 1987). This allows the analysis and verification of Petri net solution proposals by model checking (Clarke et al., 1986; Damm et al., 1990).
Figure 1 is a snapshot of some of the environments of PETRI-HELP. The upper left shows the temporal logic specification to the modelling task "Restaurant". Initially, the waiter is sleeping (starting condition: Ws). The window on the upper right of Figure 1 explains the abbreviations used in the formulas.

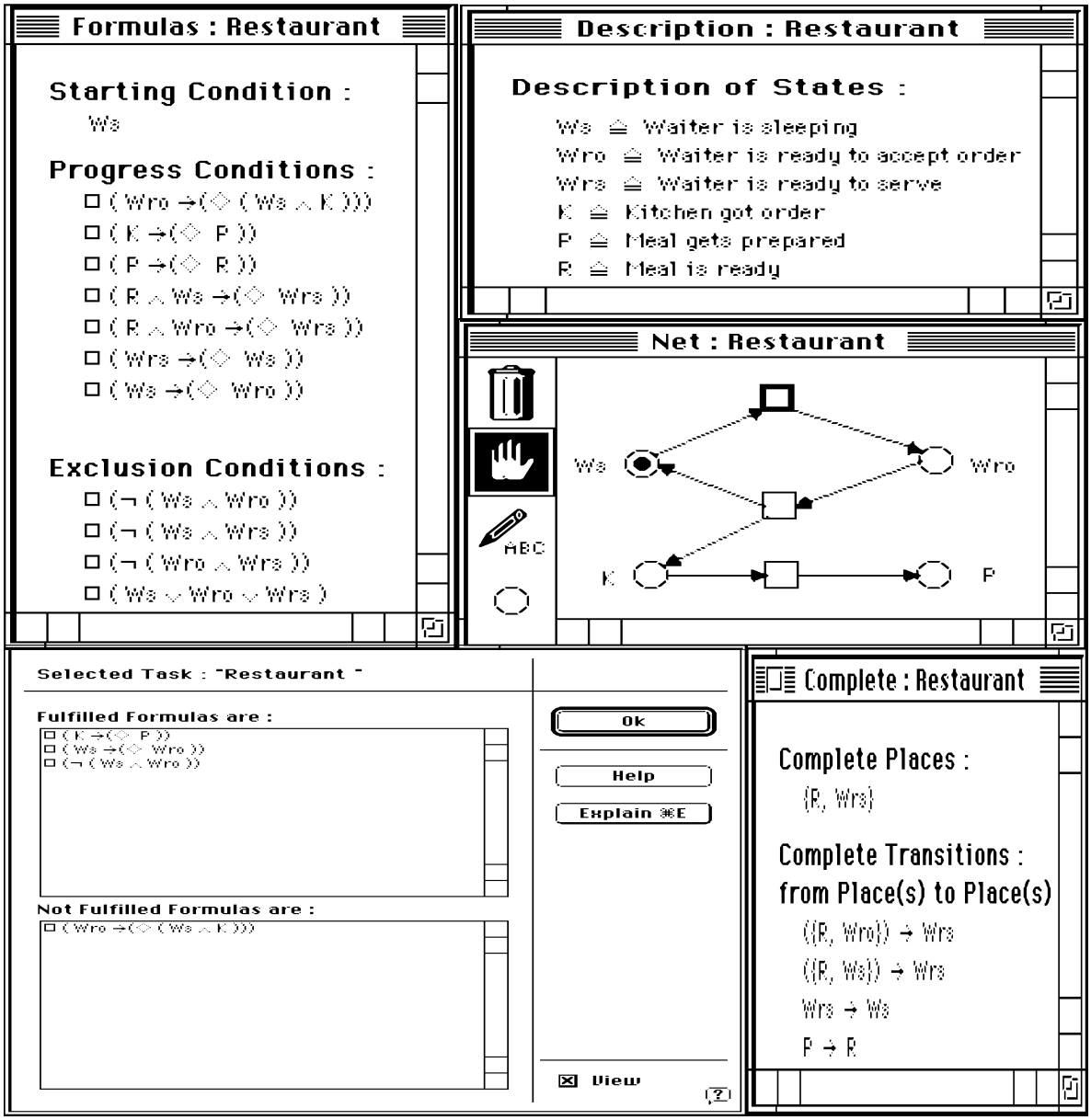

Figure 1: A snapshot of the environments of PETRI-HELP

The temporal logic formulas consist of *progress conditions* specifying time relations between states, and *exclusion conditions* specifying that certain states cannot be present (or absent) at the same time. **O**, ◊, **O** are the temporal logic operators. Informally, **O** means "always" ("it is always true that ..."), ◊ means "eventually" ("now or at some point in future it will be true that ..."), and **O** means "nexttime" ("at any next point in time it will be true that ..."). So for example **O(Wro → (◊ (Ws ∧ K)))** means: "It is always true that if the waiter is ready to accept an order then the waiter will eventually be sleeping, and the kitchen has the order."

The window "Net: Restaurant" of Figure 1 depicts a snapshot of the PETRI-HELP net editor: a condition-event Petri net solution proposal to the "Restaurant" task. A condition-event net consists of a set of places P representing states (depicted as circles), a set of transitions T representing events or processes (depicted as rectangles), and a set of directed edges (depicted as arrows) connecting places with transitions and vice versa. A place may contain a token indicating that the state represented by the place is actually true at a given time. A transition t is able to fire iff every place of its preset (i.e., the set of places p with an arrow pointing to t) contains a token and every place of its postset (i.e., the set of places q where an arrow from t points to) is empty. After firing a transition, each place of its preset is empty (i.e., the corresponding states are not true any more), and each place of its postset contains a token (the corresponding states are true now). In PETRI-HELP, transitions able to fire are marked boldly. So in Figure 1, the transition with the preset {Ws} and the postset {Wro} is able to fire since "Ws" contains a token and "Wro" is empty.
When the learner is constructing a Petri net to a given task, he or she may state hypotheses about which subset of the formulas is fulfilled by the actual state of the solution. Hypotheses are stated by simply marking the respective task formulas. The system then analyzes the selected formulas (the *hypothesis*) by model checking. As the result, it returns the formulas fulfilled and not fulfilled by the current state of the solution (see the window "Selected Task: Restaurant" on the lower left of Figure 1). In model checking, the formulas are interpreted on the case graph of the Petri net proposal. Each node of a case graph represents a possible state of the net, i.e., a set of places containing tokens at the same time. An arc represents the transition from one set of places containing tokens to another set. For example, when analyzing the formula $\mathbf{O(K \rightarrow (\Diamond P))}$ the model checker verifies whether from each node of the case graph of the Petri net containing "K", a node containing "P" is reachable.
In addition, the learner may receive why-not explanations for the unfulfilled formulas. For example, the fact that $\mathbf{(Wro \rightarrow (\Diamond (Ws \wedge K)))}$ is not fulfilled by the net in the window "Net: Restaurant" is explained by showing that a state is reachable where "Wro" has a token, but "Ws" has not, and no transition is able to fire.
PETRI-HELP also offers completion and correction proposals. The lower right of Figure 1 shows a completion proposal: two places and four transitions. These proposals are based on rules the system learns from its users. These rules associate i) successive stages of a Petri net solution proposal, or ii) Petri net fragments and the formulas fulfilled by them.

3. A cognitive model of the process of creating Petri nets

In the first subsection, we will state the kinds of hypothetical design knowledge to model distributed systems with Petri nets. Then the runnable model will be described.

3.1 A taxonomy of design knowledge

Figure 2 depicts the hypothetical design knowledge for modelling distributed systems with Petri nets. It is a summary of ISP-DL guided protocol analyses of single subjects working with PETRI-HELP.

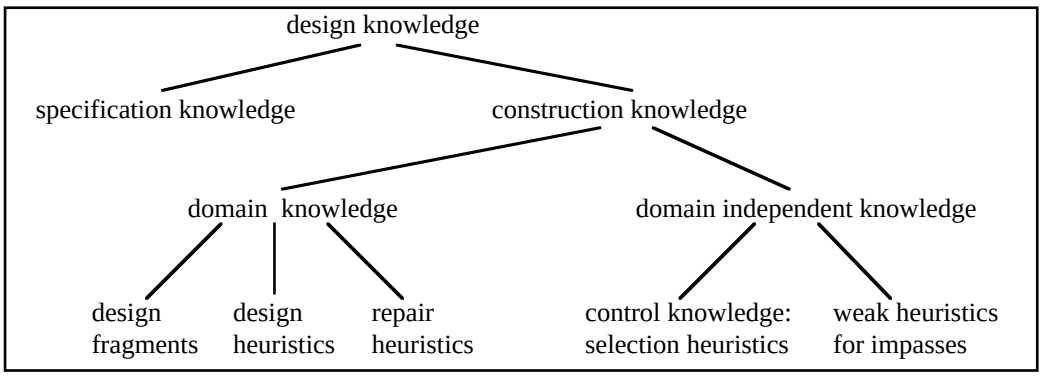

Figure 2: A taxonomy of design knowledge for modelling distributed systems with Petri nets

Specification knowledge is the knowledge utilized in specifying a distributed system. PETRI-HELP has a component supporting the specification process, but it will not be considered here.
Construction knowledge is the knowledge to construct a runnable model of a distributed system (i.e., a Petri net) from a specification. It consists of *domain knowledge* and *domain independent knowledge*. The domain knowledge consists of i) *design fragments* which relate pieces of temporal logic formulas to Petri net fragments, ii) *design heuristics* relating temporal logic progress conditions to Petri net fragments (they can be viewed as the

result of chunking of design fragments, thus they represent optimized knowledge), and iii) *repair heuristics* coming into play if design fragments and design heuristics turn out to be not sufficient for constructing a net solution. The domain independent knowledge consists of *control knowledge*, specifically *selection heuristics* for selecting the next formula to work on, or for moving to the next problem solving phase (deliberating, planning, executing, evaluating), and *weak heuristics* utilized at impasse situations, like looking for help information.
Figure 3 shows some design fragments and corresponding design heuristics. The four design fragments on the upper left of Figure 3 describe the stepwise construction of a net fragment for the progress condition schema $\mathbf{O(X \rightarrow \Diamond\ Y)}$. The corresponding design heuristic represents a chunk relating this progess condition to the result of the four steps on the left. The name *heuristic* indicates that it does not necessarily lead to a solution. The design fragments and heuristics were obtained from video-taped single-subject studies. For example, if a subject tests hypothesis H_i and later tests hypothesis H_{i+1}, and H_{i+1} differs from H_i by one additional formula F, and between these two tests the subject creates the net fragment N, then it seems reasonable that the subject considers N to be a realization of F. This assumption was supported by the verbal comments we obtained from our subjects. The F-N-associations obtained in this way correspond to the design heuristics.
We also extracted design fragments for more complex progress condition schemata containing conjunctions and/or disjunctions. The design fragments in the lower left part of Figure 3 can be viewed as the result of *generalizations* of design fragments of the upper left. The design heuristics are related by *embeddability*. Thus, the net fragment of the upper design heuristic in Figure 3 is part of the lower one.
Design fragments or design heuristics may not always be sufficient to create a solution that satisfies a given specification. If after applying them for each progress condition, there are still unsatisfied formulas, then *repair heuristics* come into play. Based on protocol analyses, the following repair heuristics were identified:

• *Heuristics for unfulfilled progress conditions*. Progress conditions may be unfulfilled for two reasons:
Firstly, the places representing the premise may not be reachable. The heuristic tries to find another formula with the same premise. If such a formula is found, the corresponding Petri net fragment is identified, and edges are created that lead from the transition of this net fragment to the places representing the premise in question. So these places will receive tokens again. The upper part of Figure 4 shows an example. Suppose the unfulfilled formula is $\mathbf{O(X \rightarrow \Diamond\ Z)}$. After the transition with preset {W, X} and postset {Y} has fired, the place X is not reachable any more. The heuristic mends this situation by proposing the bold edge on the right.
Secondly, the places representing the conclusion of the progress condition may not be reachable because of a deadlock (no transition is able to fire). In this case the heuristic identifies the places containing tokens in the deadlock situation. Then edges are created from these places to the transitions which postsets contain the places representing the premise of the progress condition. The lower part of Figure 4 shows an example: The formula $\mathbf{O((W \wedge X) \rightarrow (\Diamond\ (Y \wedge Z)))}$ is not fulfilled: When the transition with postset {W, X} fires, the premise of the formula holds. Now the transition with the preset {W, X} and postset {Y, Z} fires, leading to tokens on Y and Z. If now the transition with preset {Z} fires, V receives a token. Again the transition with postset {W, X} may fire so the premise of the formula is true. But now the transition with preset {W, X} and postset {Y, Z} cannot fire since the place Y is not empty. The conclusio of the formula cannot be fulfilled, thus the formula is not true. A deadlock has occurred. The heuristic mends this situation by proposing the bold edges on the right.

• *Heuristics for unfulfilled exclusion conditions*. If an exclusion condition schema $\mathbf{O\neg\ (X \wedge Y)}$ is not satisfied, then for each transition having **Y** in its postset, an edge is created from **X** to this transition, and vice versa (Figure 5, upper part). Thus X can only receive a token if Y loses its token, and vice versa. If a formula $\mathbf{O(X\ v\ Y)}$ is not satisfied (thus excluding the *absence* of both X and Y), then edges are created such that the transitions t with **X** in their postset have **Y** in their preset. The transitions with **Y** in their preset are removed, and edges are created from the transitions t to the places in the postset of the just removed transition. This is also done vice versa (Figure 5, lower part). Thus X can only lose its token if Y receives one, and vice versa.

3.2 The model

The model is generic in that it is intended to represent the domain dependent and domain independent knowledge, heuristics, and knowledge acquisition and modification processes of Petri net modellers of varying expertise. Thus it is supposed to model novices as well as experts. The problem solving and knowledge acquisition processes of specific Petri net modellers are viewed as instantiations of the model. Figure 6 gives an overview.
The model is structured according to the ISP-DL Theory. A run starts with the presentation of a *task*: a temporal logic specification of a distributed system. The first action ("deliberate") consists of *selecting* a formula to work on. We extracted different strategies from the protocols: selecting formulas in the order they are stated, selecting the formula with the minimum number of logical operators, or selecting a formula that can be handled by a design heuristic. When a formula is selected, a *goal* is set to implement it. A Petri net fragment for this formula has to be *planned* using the available knowledge: design heuristics, design fragments, or repair heuristics.

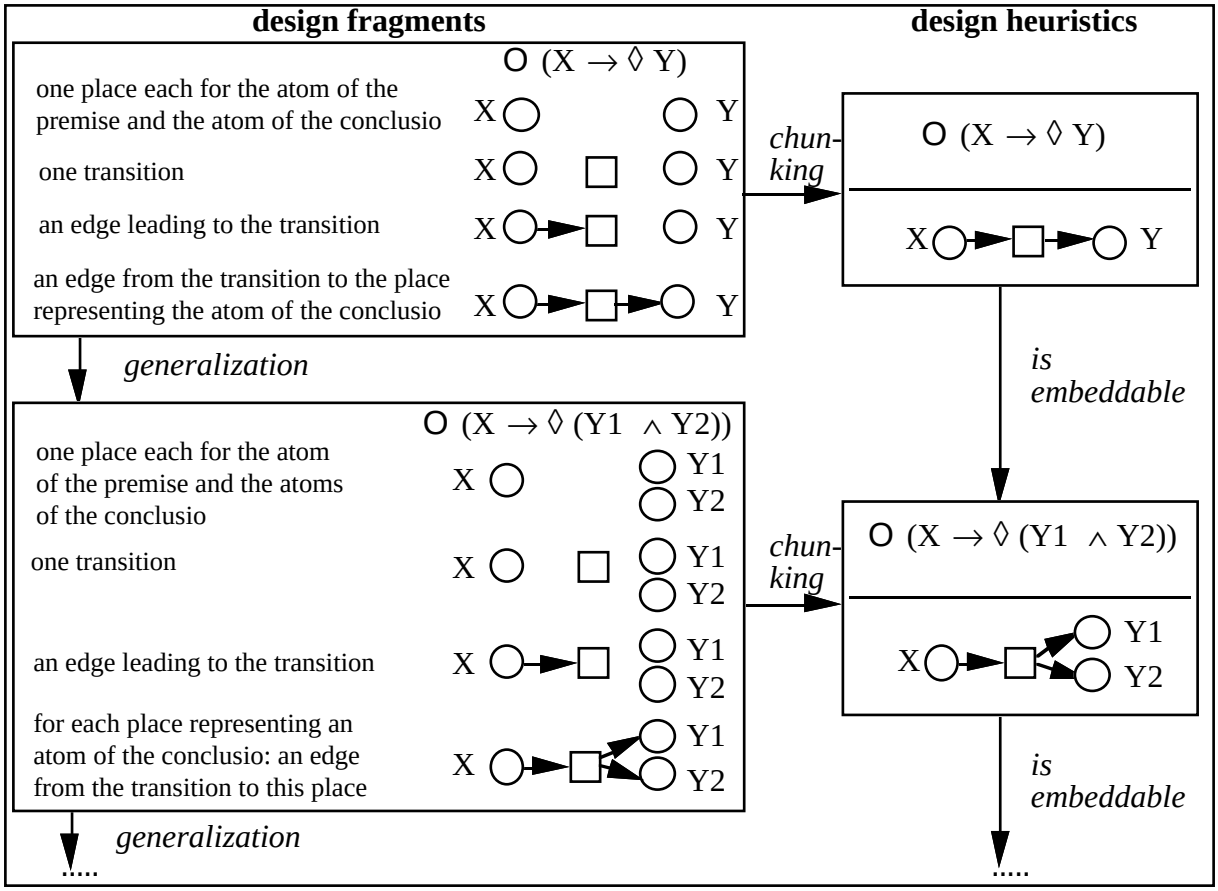

Figure 3: Design fragments and design heuristics relating formulas to net fragments

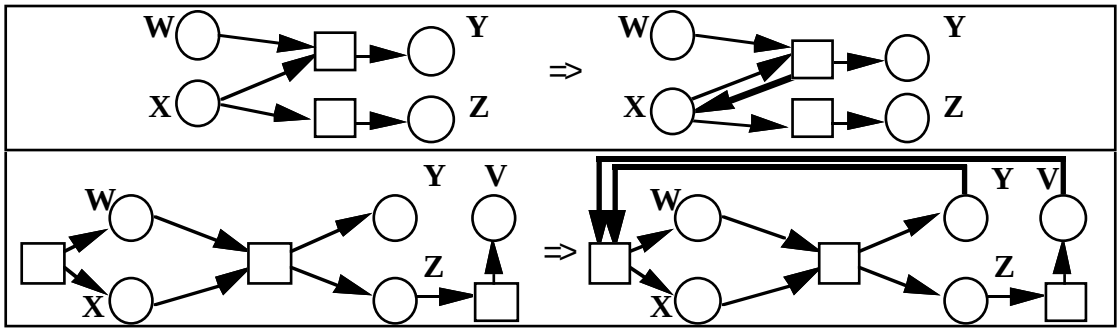

Figure 4: Two repair heuristics for unfulfilled progress conditions

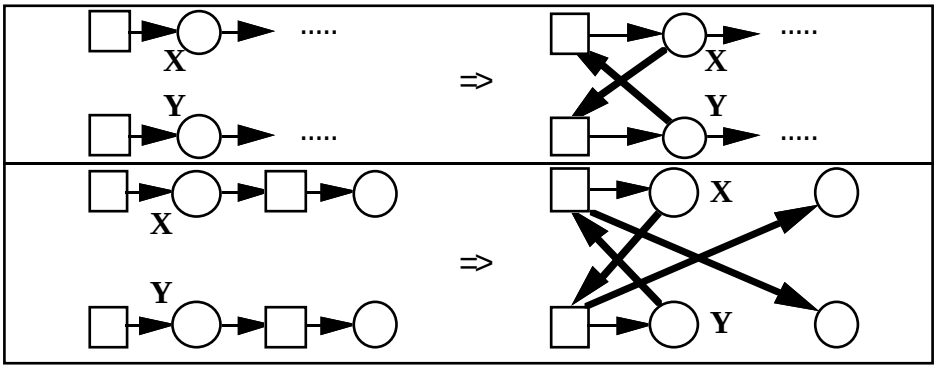

Figure 5: Two repair heuristics for unfulfilled exclusion conditions

The plan is *executed*, leading to a Petri net fragment ("*protocol*") which is *evaluated* by hypotheses testing or by simulation. If the formula is fulfilled ("*reaction to success*") then the goal is set to implement another formula. If a *solution* is reached, the knowledge used is *optimized* (design fragments are chunked, leading to design heuristics, see Figure 3). There are two sources of *impasses* in the model: Planning might fail because of *missing domain knowledge*, and evaluation might reveal that a *formula is not fulfilled*. In both cases, a *subgoal* is created to overcome the impasse by *weak heuristics*. This is represented by a recursive call of "problem solving":

• If the impasse arose because of missing domain knowledge, a generalization of the design fragments (see Figure 3) is tried. If this solves the impasse, *new knowledge* (i.e. the generalized design fragment) is acquired. If not, another formula is selected, or completion / correction proposals are asked for.

• If the impasse arose because of an unfulfilled formula, the repair heuristics described above are considered. An applicable heuristic is applied, and it is checked whether the previously unfulfilled formula is fulfilled. If so, then a rule is created that associates the net where the impasse arose with the actual net where the impasse is solved (acquisition of new knowledge). If not, hypotheses are tested, and again completions/corrections are asked for.

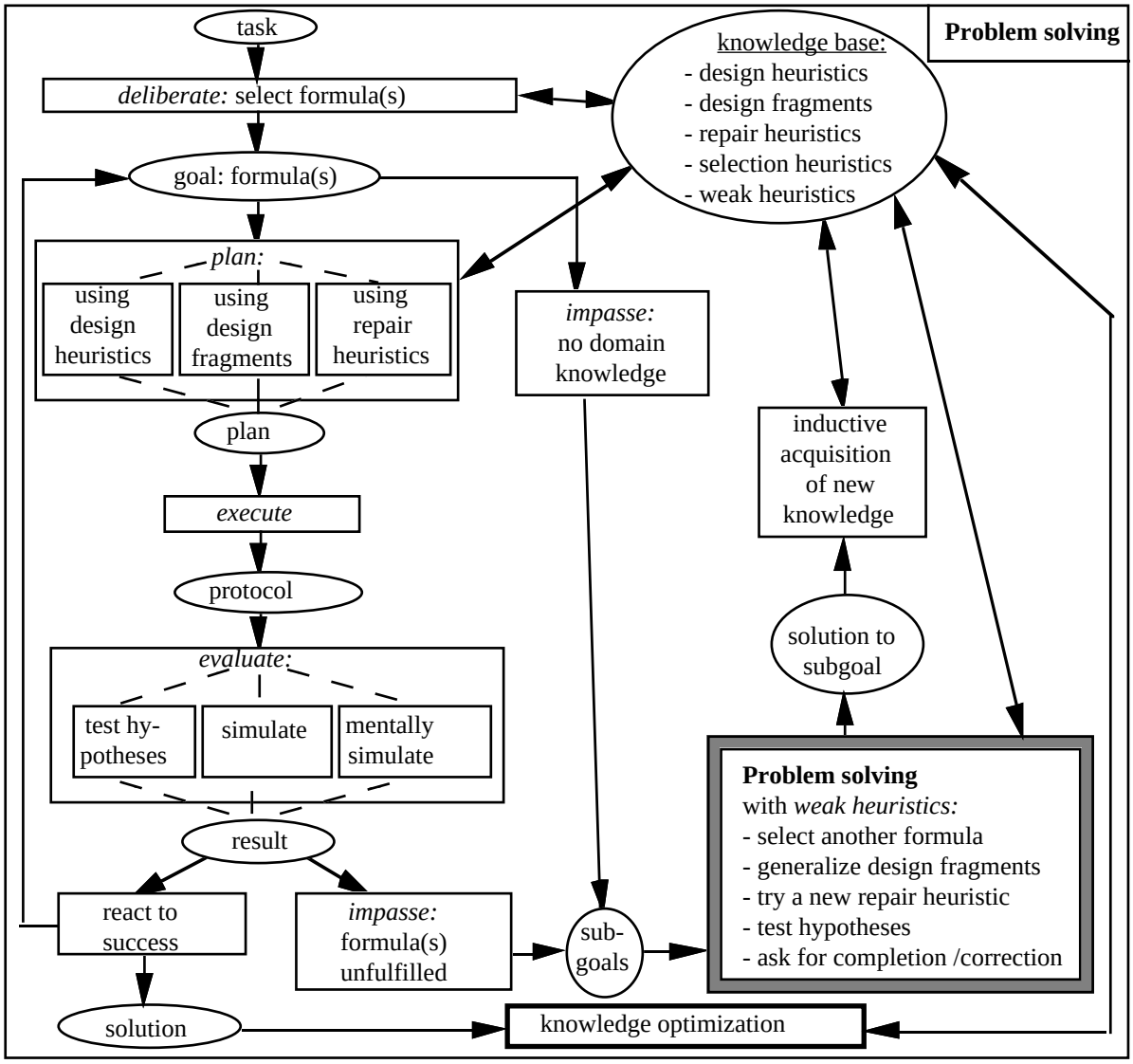

Figure 6: The model as a higher-order Petri net

4. Conclusions

We presented a model of problem solving, knowledge acquisition, and knowledge optimization in the domain of modelling distributed systems with condition-event Petri nets, a domain without a worked-out design theory. The

model attempts to describe these processes by simple, weak heuristics within the general framework of the ISP-DL Theory. The heuristics should be of interest also to other areas of design problems.
The model seems to provide a feasible basis for *dynamically diagnosing* the changing knowledge states of Petri net modellers online, and to supply appropriate feedback and help information (see Möbus et al., 1994, for the domain of functional programming). Help information would consist of i) proposing the actually needed design fragments and design heuristics, and by motivating them by showing how they can be derived from the already acquired knowledge by generalization or chunking, and of ii) proposing repair heuristics and explaining how they resolve the actual impasse. A necessary condition for dynamic diagnosis and adaptive help is to define empirical indicators for the hypothetical knowledge, heuristics, and impasses that can be registered automatically. For example, a reasonable indicator for design *fragments* is the piecemeal construction of places, transitions, and edges, interrupted by cursor movements and by pointing to the formula to be implemented. In contrast, a design *heuristic* should be implemeted fast and uninterruptedly (see also Möbus et al., 1994).
Since the model is not yet fully implemented and integrated into PETRI-HELP, it has not been empirically tested. This will involve testing predictions derived from the model's use for dynamic diagnosis, and testing the effectiveness and acceptance of the help information. A preliminary analysis of 171 solutions of subjects working with PETRI-HELP showed that 153 (89,5%) could be explained by the heuristics stated by the model.
The model is not yet complete. Heuristics not yet handled by it are: realizing more than one formula at a time (thus applying large chunks), and trying to be parsimonious, i.e., deliberately not implementing all formulas.
Currently we work on extending PETRI-HELP to other domains. For example, in the domain of hydraulic circuits, task specifications consist of function diagrams (which can be transformed into temporal logic formulas), and the solutions consist of hydraulic circuits which may be translated into Petri nets or case graphs. Problem solving in this area should be similar in many respects to problem solving in PETRI-HELP. Hypotheses about problem solving and knowledge acquisition could be utilized in this domain as well.

5. References

Anderson, J.R. (1989). A theory of the origins of human knowledge. *Artificial Intelligence*, 40, 313-351.

Clarke, E.M., Emerson, F.A. & Sistla, A.P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2), 244-263.

Damm, W., Döhmen, G., Gerstner, V., Josko, B. (1990). Modular verification of Petri nets. The temporal logic approach. In J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds): *Proceedings REX-Workshop on stepwise refinement of distributed systems: models, formalisms, correctness*. Springer (LNCS 430).

Gollwitzer, P.M. (1990). Action phases and mind-sets. In E.T. Higgins & R.M. Sorrentino (eds): *Handbook of motivation and cognition: Foundations of social behavior*, Vol.2, 53-92.

Kröger, F. (1987). *Temporal Logic of Programs*. Berlin: Springer.

Laird, J.E., Rosenbloom, P.S., Newell, A. (1986). *Universal subgoaling and chunking*. Boston: Kluwer.

Möbus, C. (1995). Towards an Epistemology of Intelligent Problem Solving Environments: The Hypothesis Testing Approach. This volume.

Möbus, C., Pitschke, K., Schröder, O. (1992). Towards the theory-guided design of help systems for programming and modelling tasks. In C. Frasson, G. Gauthier & G.I. McCalla (eds): *Intelligent tutoring systems, Proceedings ITS 92*. Berlin: Springer, LNCS 608, 294-301.

Möbus, C., Schröder, O., Thole, H.-J. (1994). Diagnosing and evaluating the acquisition process of programming schemata. In J.E. Greer, G. McCalla (eds): *Student Modelling: The Key to Individualized Knowledge-Based Instruction*. Berlin: Springer (NATO ASI Series F: Computer and Systems Sciences, Vol. 125), 211-264.

Möbus, C., Schröder, O., Thole, H.-J. (1995). Online Modelling the Novice-Expert Shift in Programming Skills on a Rule-Schema-Case Partial Order. In K.F. Wender, F. Schmalhofer, H.-D. Böcker (eds): *Cognition and Computer Programming*. Norwood: Ablex, 63-105.

Newell, A. (1990). *Unified theories of cognition*. Cambridge: Harward University Press.

Olderog, E.-R. (1991). *Nets, terms, and formulas*. Cambridge: Cambridge University Press.

Pitschke, K. (1994). User modelling for domains without explicit design theories. In *Proceedings of the Fourth International Conference on User Modelling (UM94)*. Hyannis, MA, USA, The MITRE Corporation, 191-195.

Reisig, W. (1985). *Petri nets - an introduction*. Berlin: Springer.

Reisig, W. (1992). *A primer in Petri net design*. Berlin: Springer.

Schröder, O., Möbus, C., Pitschke, K. (1993). Design of help for viewpoint centered planning of Petri nets. In B. Brna, S. Ohlsson, H. Pain (eds): *Proceedings of the 5th International Conference on Artificial Intelligence and Education (AI-ED 93)*. Association of the Advancement of Computing in Education (AACE), 370 - 377.

Van Lehn, K. (1991). Rule acquisition events in the discovery of problem solving strategies. *Cogn. Science*, 15, 1-47.

6. Acknowledgements

We thank JÖRG FOLCKERS for implementing the PETRI-HELP interface.
This research was supported by the Stiftung Volkswagenwerk (Az. 210-70631/9-13-14/89)