

Formalization and Validation

An Iterative Process in Model Synthesis

Jörg Desel

Katholische Universität Eichstätt-Ingolstadt
Lehrstuhl für Angewandte Informatik
85071 Eichstätt, Germany
joerg.desel@ku-eichstaett.de

ABSTRACT. This work considers model synthesis and validation in controller design. The problem we are interested in is to derive a formal model of a controlled automation system from a semi-formal description of the uncontrolled plant and various requirements concerning the plant and the processes of the controlled system. These requirements are formulated on many different abstraction levels, partly employing formal notations, partly using just natural language and partly consisting of mixtures of both. Moreover, they are often incomplete, contain errors, contradict each other and assume some domain knowledge which is typically not explicitly stated. So a crucial part of the model synthesis process is the formalization of the plant and of the requirements as well as validation of the derived models. We suggest a simulation-based method which employs formal and graphical representations of process models and specifications and which involves an iterative process of formalization and validation of requirements. The approach uses Petri nets as formal process models and partially ordered runs as their semantics. This contribution also reports on experiences with applying the method for an industrial case study and on an according developed tool.

1 Introduction

This contribution is on model based development of software systems that are supposed to run in a technical environment. More precisely, we deal with the development of such systems which is based on formal process models. We formulate a view from computer science (or *informatics*, as we call this discipline in Germany). However, the concepts described were developed together with automation engineers from academia and from industry.

Model based system development can only lead to a valuable system if the underlying models faithfully represent the requirements. The requirements include information about the existing or the planned environment of the system as well as the desired system behavior within this environment. These statements hold true for a wide range of systems. In this work we concentrate on *computer* systems which are supposed to function in a given *technical* environment. These include automation systems composed of a plant and a control restricting the plant's behavior. In this setting, the aim is to develop a control algorithm such that the controlled system matches the requirements. Unfortunately, the requirement specification is often formulated on many different abstraction levels, partly employing formal notations, partly using just natural language and

partly consisting of mixtures of both. Moreover, it is usually incomplete, contains errors, is contradictory and assumes some domain knowledge which is not explicitly stated.

Since the general aim is to develop the controller software, one possible approach would be to start with generating a formal specification of this software. This software has to run within the environment. Therefore, a formal specification of this environment, namely the plant, is necessary as well. This specification is not easy to obtain because the user is interested in the overall behavior. Thus he will only provide information concerning the controlled system, i.e. the composition of plant and control. Moreover, the precise behavior of the plant might be unknown as well. Faulty assumptions on the plant specification will lead to faulty or incomplete control specifications, which eventually leads to controller software that matches the specification but does not satisfy the user's needs.

Therefore, we proceed differently; we aim at a model of the entire system, including both the plant and the control. This model can be viewed as a *specification* of the total system. A given control software matches the specification if its behavior together with the plant precisely corresponds to the behavior of the model. The model of the entire system is generated from the different specification items that are given in different form mentioned above. The crucial steps in model synthesis are the appropriate formalization of the requirements (and their validation) and the correct synthesis of the model from the formal specifications.

This work will present an approach for model synthesis for controlled systems that employs different formalization / validation steps and a synthesis procedure to obtain the model from the specifications in a systematic way. The approach is based on Petri nets and the simulation of Petri net models. By simulation we mean construction and inspection of causal runs, represented again by Petri nets.

Model synthesis is used in controller design, for the examination of specifications w.r.t. feasibility and for creation of reference models for the final system that are used for verification and tests. These models are also very useful as a basis for model-based test case generation. So we view model synthesis, formalization and validation as *one* important early phase in system development.

The paper is organized as follows: In the forthcoming section we describe what we mean by validation of models, in contrast to system validation. We also distinguish validation from verification and formalization from specification. Section three is devoted to the steps of our approach in a general setting. In section four, the formal model, namely Petri nets and its causal semantics, is presented. The causal simulation of Petri nets, its advantages and some words about algorithmic aspects is the topic of section five. Whereas the forth and fifth sections only contain an academic example, section six reports on experiences with this approach obtained with an industrial case study. This section also contains extensions of the modeling language and conclusions.

The first sections of this paper are strongly based on [De00a] and [De02], where more details can be found. Different aspects of the approach were also adapted to and presented in various different communities (see [De99, De00b], and see [De00c] for using part of the concept for education purpose).

2 Model Validation

This section is devoted to a general discussion of the term “model validation” in system design. Validation is usually related to systems. We adopt its meaning to models.

The usual definition of validation of a *system* in relation to verification and evaluation reads as follows:

Validation. Validation is the process determining that the system fulfills the purpose for which it was intended. So it should provide an answer to the question “*Did we build the right system?*” In the negative case, validation should point out which aspects are not captured or any other mismatch between the system and the actual requirements.

Verification. Verification is the automated or manual creation of a proof showing that the system matches the specification. A corresponding question is “*Did we build the system right?*” In the negative case, verification should point out which part of the specification is not satisfied and possibly give hints why this is the case, for example by providing counter examples. Nowadays, *model checking* is the most prominent technique used for automated verification. *Proof techniques* can be viewed as manual verification methods.

Evaluation. Evaluation concerns the questions “*Is the system useful?*”, “*Will the system be accepted by the intended users?*” It considers those aspects of the system within its intended environment that are not formulated or cannot be formulated in terms of formal requirements specifications. The question “*How is the performance of the system?*” might also belong to this category, if the system’s performance is not a matter of specification.

This contribution is about validation of *models*, namely process models. So replacing the term “system” in the above definitions by “process model” should provide the definitions we need. Models are used as specifications of systems. Unfortunately, replacing “system” by “specification” in the definitions does not make much sense. So we need a more detailed investigation of the role of models and of validation in model-based system development.

The following figure presents the usual view:

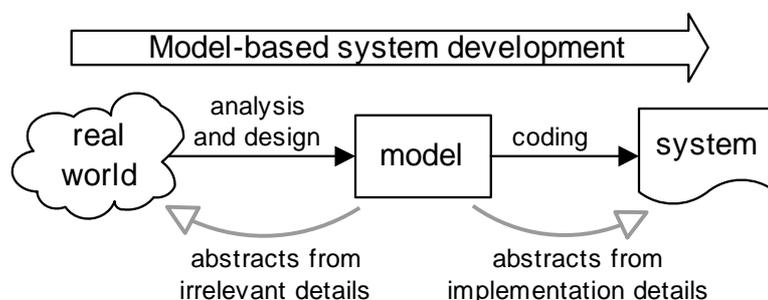


Fig. 1: Model based system development

In this figure, the model is an abstract representation of both, the relevant part of the “real world” and the actual system implementation. It abstracts from irrelevant details of the considered part

of the “real world”, and it abstracts from implementation details of the system. Verification mainly concerns the relation between the model and the system implementation, validation concerns the relation between the model and the “real world”, whereas evaluation directly relates the system and the “real world”.

The above view ignores that the system to be implemented will have to function within an environment, which also belongs to the “real world”. So the left hand side and the right hand side of the picture cannot be completely separated; they are linked via the “real world”. The following figure shows a more faithful representation of the situation.

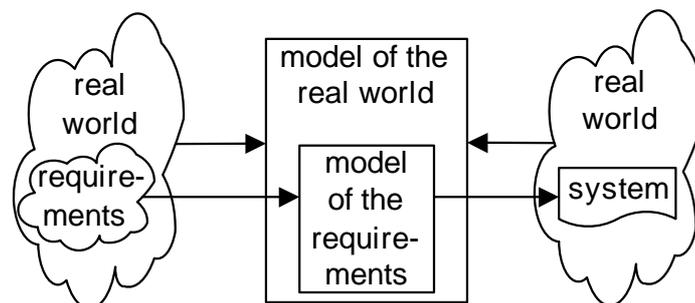


Fig. 2: Capturing the embedding in the real world

Notice that the word “system” is used with different meanings: the “real world” (environment plant), the software system to be implemented (control) and the composition of both (the controlled plant). In the sequel we mainly use the term for the environment together with (part of) the control.

A more detailed view of the model distinguishes *requirements specification* and *design specifications* on the level of the model.

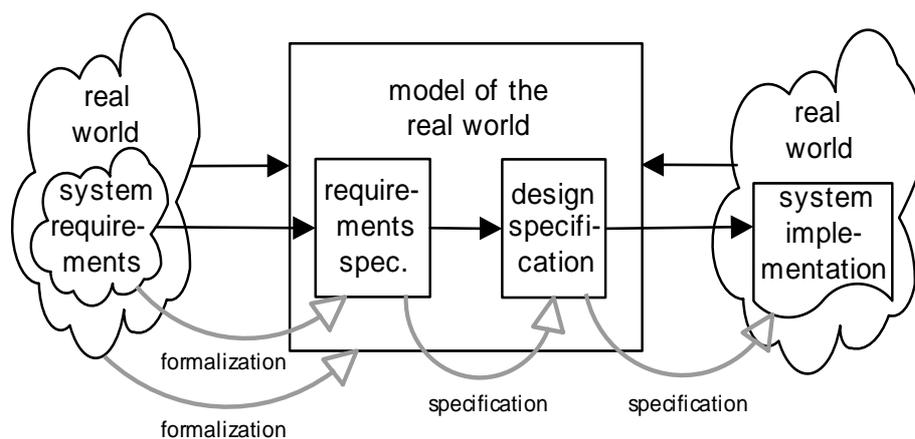


Fig. 3: Capturing requirements and design specifications

The model of the real world is obtained by analysis of the domain and *formalization* of its relevant aspects. The requirements specification models the requirements and is derived by *formalization* of the requirements that exist within the “real world”. The design specification can be

viewed as a model of the system implementation, without considering implementation details though. This model has to satisfy all properties formulated in the requirements specification. The transformation from the requirements specification to the design specification is a nontrivial task. Finally, there should be a more or less direct transformation from the design specification to the system implementation. This implementation of the system is also said to be *specified* by the design specification.

Now let us consider the reverse direction. It is a matter of *verification* to check whether the design specification actually matches the requirement specification. It can also be *verified* whether the system implementation reflects the design specification. The correctness of the formalization transformations can only be checked by *validation*. So “formalization” and “validation” is a related pair of terms in the same sense as “specification” and “verification”. Finally, requirements that are not captured in the model can only be checked by *evaluation* of the system implementation within the “real world”.

In the following figure, the arrow annotated by “evaluation” points to the “real world” including the system requirements whereas the lower arrow annotated by “validation” addresses only the “real world” without system requirements.

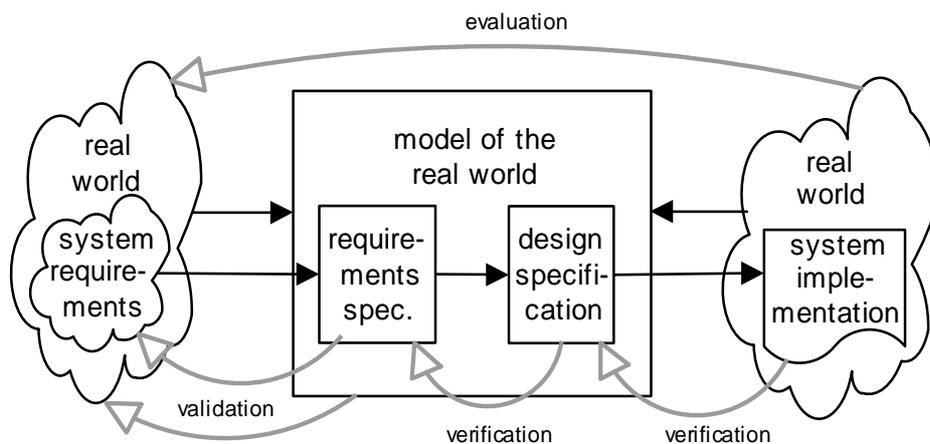


Fig. 4: The position of validation, verification and evaluation

In our context of controller design, the plant is part of the real world (the environment, respectively) and the control plays the role of the system implementation. Formalizing the description of the plant will yield a formal process model whereas the formalization of the requirements have to be interpreted on this process model, or, respectively, on its behavior. Both formalization steps have corresponding validation steps that are supported in our approach.

3 The Approach

How can we derive a formal model from a semi-formal description of a controlled system and of its desired behavior? There is no general answer to this question, since modeling is a creative process. Creating a model always means to formalize concepts that have not been formulated

that precise before. Therefore, misunderstandings, errors, missing assumptions etc. can not be avoided in general. The best we can expect is to provide means for detecting these errors as soon as possible.

We concentrate on process models that have a dynamic behavior and can thus be executed. So for each process model there is the notion of a run, i.e., one of its executions. Our basic assumption is that the domain expert (the user of our approach) knows well what the correct runs of the desired system should look like but might have problems in formalizing an appropriate specification of this set of runs. We will use *causal* runs, given by partially ordered sets of events and local system states. A precise definition of causal runs and their graphical representation is deferred to the next sections.

As mentioned in the previous section, formalization tasks appear at different steps: First, a given or planned system that serves as the environment or plant has to be modeled. Second, the requirements of the controlled system has to be specified. Both aspects deserve additional validation procedures. Given a valid model of the plant and a valid specification of the controlled system, the following step is to design the control algorithm and to verify its correctness with respect to the specification. This step is not within the scope of this contribution. However, it will turn out that some verification means can also be used for validation purposes.

We first consider the problem of modeling a given system (the environment). The behavior of the system should precisely correspond to the behavior of the model. Assuming that we have a version of this model, our approach generates the runs of the model, visualizes this behavior in an appropriate way and presents the result to the expert. This model is often derived directly from the system's structure and architecture. If the behavior of the system rather than its structure is known, then a first version of the system model is constructed from the runs by *folding* appropriate representations of runs (this procedure is given in [DE00b] for workflow models).

The simulation of the system model either shows that the model can be accepted or that it does not yet match the system. In the latter case, the model is changed according to identified modeling errors and the procedure is repeated. Only when the simulated runs of the model coincide with the required runs, the model can be used to obtain information about the system. The procedure for model validation can be complemented by verification means: If some behavioral properties of the system are known then the model should satisfy according properties as well. Since this verification step is sometimes hard to conduct, there is an intermediate solution for properties that all runs should satisfy: Simulation is paired with verification of the simulated runs. This requires an analysis method for runs, which is also the kernel of the formalization of other requirements, to be discussed next.

Now we consider the formalization and validation of requirements. That is, we assume to have a valid model of the environment (the plant) and add requirements that have to be satisfied by the controlled system, i.e., that have to be guaranteed by the desired control. In our approach, we only consider required properties that can be formulated as properties of runs (generally, all properties of a Linear Time Temporal Logic). These requirements are formalized, validated and implemented step by step. In the first step, we begin with some of the requirements and analyze simulated runs of the existing model with respect to these requirements. The result is a distinction of those runs that satisfy the requirements and those that do not. This way the user gets in-

formation about his requirement specification in terms of runs (“did you really want to rule out precisely those runs that failed the test?”). Figure 5 illustrates this step.

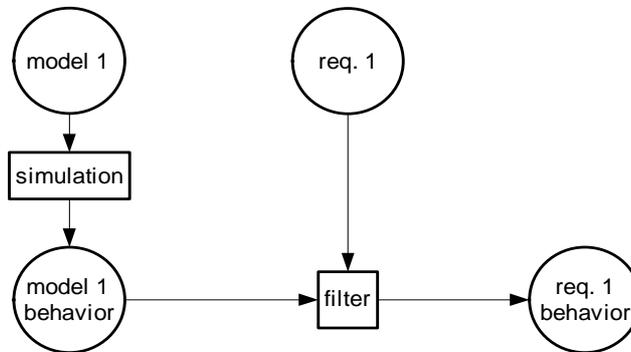


Fig. 5: A first step in requirements validation

After an iterative reformulation of the first requirements the simulation based approach should eventually yield a valid specification of this requirement. Thereafter the system is modified in such a way that it satisfies this requirement. For some requirement specifications, there is an automated procedure for this task. In general, however, there is some freedom in how to implement the requirement. The implementation of the requirement is either verified by appropriate verification techniques or checked again by simulation.

After the first step, a second requirement can be formalized, validated and implemented, based on the modified model, in the same way (see Figure 6), and so on. Notice, however, that the implementation of the new requirement should not violate a previously implemented requirement. As long as all requirements only restrict the set of possible runs, this problem does not occur. But, if *liveness* properties (requiring that something eventually happens) and *safety* properties (requiring that something bad does not happen) are added in arbitrary order, then previous verification steps might have to be repeated.

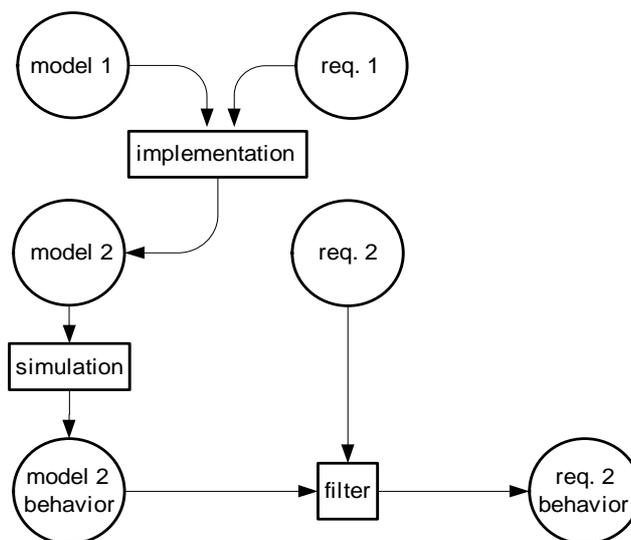


Fig. 6: A second step in requirements validation

4 Petri Nets

We concentrate on process models, i.e., on specifications of runs of a system. The formal modeling language used in our approach is given by Petri nets. In this contribution, the approach is explained by using a very simple variant of Petri nets, namely place/transition Petri nets.

Each process model has a dynamic behavior, given by its set of *runs*. In a run, *actions* of the system can occur. We will distinguish actions from *action occurrences* and call the latter *events*. In general, an action can occur more than once in a single run. Therefore, several events of a run might refer to the same action. Runs and events of Petri nets can be defined in several ways. We will discuss *sequential runs*, given by *occurrence sequences* and *causal runs*, given by *process nets*.

We roughly follow the standard definitions and notations of place/transition Petri nets and process nets [DR98, GR83]. However, in contrast to the usual notion, we equip process nets with initial states, represented by markings of conditions.

A place/transition Petri net $(P, T, pre, post, M_0)$ is given by

- a finite set P (places), represented by circles,
- a finite set T satisfying such that P and T are disjoint (transitions), represented by squares,
- two mappings $pre, post: T \times P \rightarrow \{0, 1\}$, the derived flow relation F is given by

$$F = \{(p, t) \in P \times T \mid pre(t, p) = 1\} \cup \{(t, p) \in T \times P \mid post(t, p) = 1\}$$

and is represented by arrows (notice that, conversely, F completely determines the mappings pre and $post$, which is not the case in the more general setting where the domain of these mappings is the set of nonnegative integers),

- an initial marking $M_0 : P \rightarrow \{0, 1, 2, \dots\}$ (represented by tokens in the places).

For a net element x in $P \cup T$, $\bullet x$ denotes the set of elements y satisfying $(y, x) \in F$ (the pre-set) and x^\bullet denotes the set of elements y satisfying $(x, y) \in F$ (the post-set of x).

We restrict our considerations to place/transition Petri nets without transitions t satisfying $\bullet t = \emptyset$ or $t^\bullet = \emptyset$.

Given an arbitrary marking $M : P \rightarrow \{0, 1, 2, \dots\}$, a transition t is enabled if each place p satisfies $pre(t, p) \leq M(p)$. The occurrence of t leads to a new marking M' , defined for each place p by

$$M'(p) = M(p) - pre(t, p) + post(t, p).$$

We denote the occurrence of t at the marking M by $M \rightarrow_t M'$.

Figure 7 shows a place/transition Petri net modeling a vending machine for beverages. The left hand part describes a physical facility for brewing and dispensing warm beverages. At most two warm beverages can be prepared concurrently. After dispensing a beverage, cold water is filled in the respective unit, hence the place *cold* in the post-set of the transition *dispense*. The right-hand part describes the control of the machine and a counter for coins. Initially, the machine is ready for the insertion of a coin. An inserted coin will be checked; counterfeit will be rejected. When a coin is accepted, a beverage can be dispensed and the control part of the machine returns to the state *ready*.

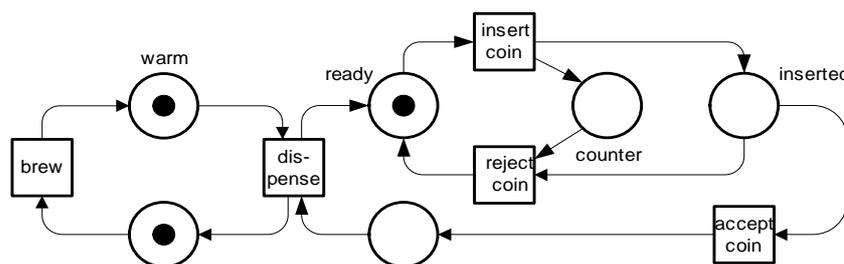


Fig. 7: A Petri net model of a vending machine

This Petri net can be viewed as a composition of a plant and a control: The left hand part as well as the transitions *insert*, *reject* and possibly the *counter* represent physical features and belong to the plant, the remaining parts belong to the control. The specification of the overall system includes that after insertion of a correct coin eventually a beverage is dispensed, otherwise the coin is rejected and nothing is dispensed.

There are basically two different techniques to describe the behavior of a Petri net model: A single run can either be represented by a sequence of action names, representing subsequent events or by a causally ordered set of events. The first technique is formally described by *occurrence sequences*. It constitutes the *sequential semantics* of a Petri net. The second technique employs *process nets* representing *causal runs*. It constitutes the *causal semantics* of a Petri net.

The main advantage of sequential semantics is formal simplicity. Sequential semantics generalizes well-known concepts of sequential systems. Every occurrence sequence can be viewed as a sequence of global system states and transformations leading from a state to a successor state. Formally, these states are not explicitly mentioned but only the transition names of subsequent events are given. One occurrence sequence of our example is

insert accept brew dispense insert accept brew dispense.

Its resulting state differs from the initial one only with respect to the place *counter*, which now carries two tokens.

One of the main advantages of causal semantics is its explicit representation of causal dependency, represented by paths of directed arcs in process nets. Consequently, concurrent events are events that are not connected by a path in a process net. Causal semantics of Petri nets has been studied in Petri net theory since a long time, starting with the work of Carl Adam Petri in the

seventies. Application projects employing Petri nets, however, mostly restrict to sequential semantics, and so do most Petri net tools.

In sequential semantics, a run is represented by a sequence of events such that causal dependencies are respected; if an event causally depends on another event, then these events will not appear in the reverse order in an occurrence sequence. A causal run also consists of a set of events, representing action occurrences of the system. An action can only occur in certain system states, i.e. its pre-conditions have to be satisfied. The occurrence of the action leads to a new system state where some post-conditions of the action start to hold. An event is therefore causally dependent on certain pre-conditions and might lead to new conditions that are causal prerequisites for other events. The time and the duration of an event has no immediate influence on the system's behavior, as long as such dependencies are not explicitly modeled as actions of clocks. Combining events with their pre- and post-conditions yields a causal run, formally represented by a *process net*. Since pre- and post-conditions of events are explicitly modeled in a process net, the immediate causal dependency is represented by the arcs of a process net. The transitive closure of this relation defines a partial order that we will call *causal order*; two events are causally ordered if and only if they are connected by a chain of directed arcs. Otherwise, they are not ordered but occur *concurrently*.

Formally, the causal behavior of a place/transition Petri net $(P, T, pre, post, M_0)$ is defined by its set of process nets, representing causal runs. For the formal definition of a process net, we employ again place/transition Petri nets: Each process net of the place/transition Petri net $(P, T, pre, post, M_0)$ is given by a place/transition net $(C, E, pre', post', S_0)$, together with mappings $\alpha : C \rightarrow P$ and $\beta : E \rightarrow T$, satisfying the conditions given below. The places of a process net are called *conditions*, the transitions *events* and the markings *states*. To avoid confusion with process nets, the place/transition Petri net model $(P, T, pre, post, M_0)$ of the system will be called *system net* in the sequel.

- Every condition c in C satisfies $| \bullet c | \leq 1$ and $| c \bullet | \leq 1$,
- the transitive closure of the flow relation K is irreflexive, i.e., it is a partial order over $C \cup E$,
- for each event e in E and for each place p in P we have

$$\begin{aligned} | \{ c \in \bullet e \mid \alpha(c) = p \} | &= pre(\beta(e), p), \\ | \{ c \in e \bullet \mid \alpha(c) = p \} | &= post(\beta(e), p). \end{aligned}$$

- $S_0(c) = 1$ for each condition c in C satisfying $\bullet c = \emptyset$ and $S_0(c) = 0$ for any other condition c ,
- $\alpha(S_0) = M_0$, where α is generalized to states S by

$$\alpha : (C \rightarrow \{0, 1, 2, \dots\}) \rightarrow (P \rightarrow \{0, 1, 2, \dots\}), \quad \alpha(S)(p) = \sum_{(\alpha(c)=p)} S(c).$$

A condition c in C represents the appearance of a token on the place $\alpha(c)$. An event e in E represents the occurrence of the transition $\beta(e)$. In a run, each token is produced by at most one transition occurrence, and it is consumed by at most one transition occurrence. Hence, conditions of

process nets are not branched. The transitive closure of K defines the *causal relation* on events and conditions. Since no two elements can be mutually causally dependent, the causal relation is a partial order. In other words, the flow relation has no cycles. Since events represent transition occurrences, the pre- and post-sets of these transitions are respected. The initial state of the process net is the characteristic mapping of the set of conditions that are minimal with respect to the causal order, i.e., these conditions carry one token each, and all other conditions are initially unmarked. Note that all minimal elements are conditions because, by our general assumption, every event has at least one pre-condition. Finally, the initial state of the process net corresponds to the initial marking of the system net, i.e., each initial token of the system net is represented by a (marked) minimal condition of the process net.

Each process net represents a single causal run of a system net. We equip a process net with an initial state S_0 so that the token game can be played for process nets as well.

The following figures show two process nets of the vending machine example of Figure 7.

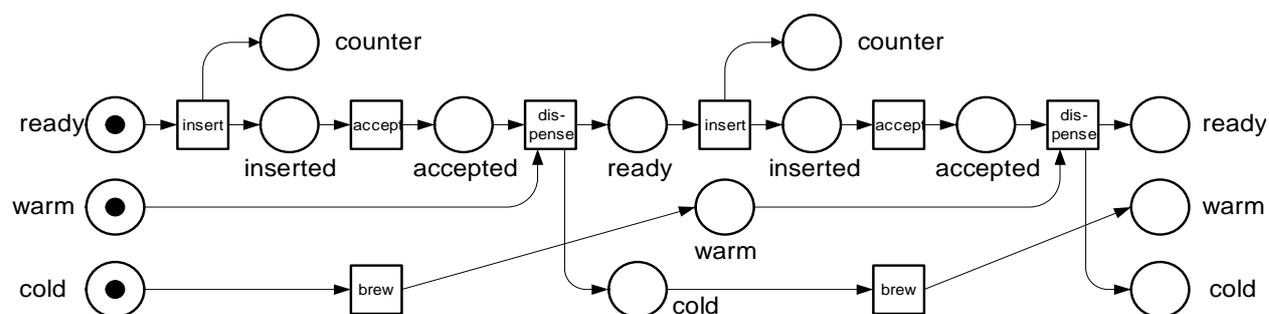


Fig. 8: A process net representing a causal run of the vending machine

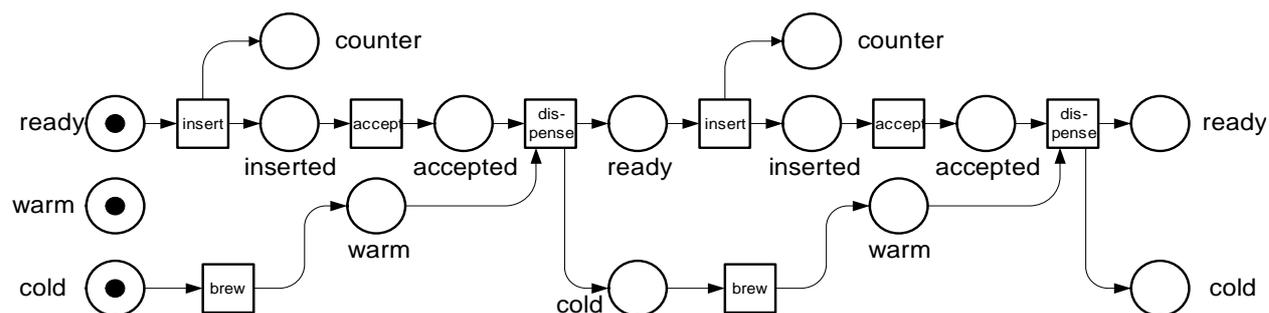


Fig. 9: Another process net

The mappings from elements of these process nets to elements of the system net are represented by annotations in the obvious way. For example, each event annotated by *insert* represents one occurrence of the transition named *insert* of the system net.

Both process nets describe causal runs where two coins are inserted, both are accepted, and two beverages are dispensed. So both runs end in state *ready*, with two tokens in *counter*, one token in *cold* and one token in *warm*. In the first run, the beverage that was warm initially is dispensed

first. Figure 8 shows a run where the second dispensed beverage is brewed concurrently to all previous events whereas in the run of Figure 9 the capacity of the brewing facility is not exploited; the second *dispense* event has to wait after the first *dispense* event at least the time necessary for brewing a new beverage. So one could consider the run of Figure 8 more efficient with respect to response time. Moreover, the quality of a warm beverage might decrease after a while (this holds at least for coffee).

Using acyclic graphs to define partially ordered runs is common for many computation models. The specific property of process nets is that each process net is formally a Petri net and that there is a close connection between a process net representing a run and the Petri net modeling the system; the events of a process net are annotated by respective names of actions of the system. More precisely, mappings from the net elements of the process net to the net elements of the Petri net representing the system formalize the relations between events of a process net and transitions of a system net and between conditions of a process net and places of a system net.

As will be stated in the following well-known lemma, sequential and causal runs have strong relations. Sequences of event occurrences of a process net closely correspond to transition sequences of the system net.

Lemma 1 [BF88]. Let $(P, T, pre, post, M_0)$ be a place/transition Petri net. If $(C, E, pre', post', S_0)$ together with mappings $\alpha : C \rightarrow P$ and $\beta : E \rightarrow T$ is a process net and

$$S_0 \rightarrow_{e1} S_1 \rightarrow_{e2} \dots \rightarrow_{en} S_n$$

is a sequence of event occurrences, then

$$M_0 \rightarrow_{\beta(e1)} \alpha(S_1) \rightarrow_{\beta(e2)} \dots \rightarrow_{\beta(en)} \alpha(S_n)$$

is a sequence of transition occurrences of $(P, T, pre, post, M_0)$. Conversely, for each sequence

$$M_0 \rightarrow_{t1} M_1 \rightarrow_{t2} \dots \rightarrow_{tn} M_n$$

of transition occurrences of the place/transition net $(P, T, pre, post, M_0)$, there is a process net $(C, E, pre', post', S_0)$ with $\alpha : C \rightarrow P$ and $\beta : E \rightarrow T$ and a sequence of event occurrences

$$S_0 \rightarrow_{e1} S_1 \rightarrow_{e2} \dots \rightarrow_{en} S_n$$

such that, for $1 \leq i \leq n$, $\alpha(S_i) = M_i$ and $\beta(e_i) = t_i$.

Therefore, roughly speaking, the set of occurrence sequences of a place/transition Petri net coincides with the set of occurrence sequences of its process nets when only the labels (β -images) of events of these latter sequences are considered.

For example, the above mentioned occurrence sequence of transitions

$$\textit{insert accept brew dispense insert accept brew dispense}$$

has related occurrence sequences of events in both process nets shown in Figures 8 and 9.

5 Simulation by Construction of Runs

By simulation we understand the generation of runs of the process model. For a valid model, each run should represent a corresponding run of the system, and for each system run there should exist a corresponding run of the model. Validation by simulation means generating and inspecting runs of the model with respect to the desired runs of the modeled system. Since neither the system nor its runs are given formally, only domain experts can do this comparison. So this task requires a good and easy understanding of the generated runs of the model.

Usually, the user is supported by a graphical representations of runs: The Petri net is represented graphically and sequential runs are depicted by subsequent occurrences of transitions of the net. We suggest to construct and visualize causal runs given by partially ordered *process nets* instead. We argue that we gain two major advantages, namely expressiveness and efficiency.

Every sequence of events, i.e. transition occurrences, defines a total order on these events. A transition can either occur after another transition because there is a causal dependency between these occurrences or the order is just an arbitrarily chosen order between concurrent transition occurrences. Hence, an occurrence sequence gives little information on the causal structure of the system run. Interesting aspects of system behavior such as the flow of control, the flow of goods, possible parallel behavior etc. are directly represented in process nets, but they are hidden in sequences of events.

Consider again the occurrence sequence *insert accept brew dispense insert accept brew dispense* of the place/transition Petri net of Figure 7. This sequence corresponds to both process nets given in Figure 8 and Figure 9. It is a sequential view of both causal runs because both process nets possess corresponding occurrence sequences. As argued before, the run of Figure 8 has reasonable advantages compared to the run of Figure 9. The above sequence does not distinguish between both runs. In particular, the sequence does not express important behavioral properties that are distinguished by causal runs. Causal runs provide full information about these causal dependencies. Interesting aspects of system behavior such as the flow of control, the flow of goods, possible parallel behavior etc. are directly represented in process nets, but they are hidden in sequences of events. For example, the process net shown in Figure 8 very clearly shows the flow of control (*ready, insert, inserted, accept, ...*) and the different local states and events of the beverages (*cold, brew, warm, dispense, ...*).

Simulation of a system model means construction of a set of (different) runs. In general, each causal run corresponds to a nonempty set of occurrence sequences. This correspondence is formally established by Lemma 1. Taking the sequence of labels of events in occurrence sequences of process nets yields all occurrence sequences of the system net. The number of event occurrence sequences of a single process net grows dramatically when a system exhibits more concurrency. Each of these occurrence sequences represents the very same causal system run. Hence, the simulation of more than one of these sequences yields no additional information on the causal behavior of the system. The gain of efficiency is most evident when all runs of a system can be simulated, i.e. when there is only a finite number of finite runs. In the case of arbitrary large runs, a set of process nets allows to represent a larger significant part of the behavior than a comparable large set of occurrence sequences.

In the example shown in Figure 8, the left event *brew* can occur at any position before the second *dispense* event. The other *brew* event can occur arbitrarily after the first *dispense*. So this process net possesses 24 maximal occurrence sequences. Some of these sequences differ only with respect to the order of the two *brew* events, but they still represent 21 different occurrence sequences of the system net.

The number of occurrence sequences of a single process net grows dramatically when a system exhibits more concurrency. In general, the number of occurrence sequences of a single process net grows exponentially with the number of concurrent transitions. Hence, the simulation of more than one of these sequences yields no additional information on the behavior of the system. However, if a system exhibits no concurrency at all, then nothing is gained by the construction of process nets because in this case each process net has only one maximal occurrence sequence.

As mentioned previously, a single occurrence sequence might correspond to more than one process net. This only happens in case of system nets that are not safe, i.e. have more than one token on a place at a reachable marking. In this case the ratio between the number of process nets and the number of occurrence sequences might be slightly reduced.

In previous publications, we have described the simulation algorithms [De00a, De02]. Crucial aspects are a compact representation of similar runs, completeness with respect to all possible alternatives and in particular termination conditions for potentially infinite runs.

As described in the third section, we have to provide means to analyze the constructed runs with respect to specified requirements. These specifications are formulated on the level of the system net in a graphical way (see [De00a]), adopting the well-known fact transitions [GT76] and introducing analogous graphical representations for other properties).

As the specifications are interpreted on runs, we developed algorithms for analysis of process nets. It turned out that the particular structure of these nets lead to significant advantages with respect to efficiency, compared to occurrence sequences, at least for some important classes of requirement specifications. As examples, we now describe two such classes, namely *goals* and *facts*.

A *goal* requires that a certain situation will eventually occur. More precisely, each process net satisfying some fairness condition should contain a condition that is mapped to a certain place. Clearly, checking whether a given process net contains such a condition is a trivial task. However, if the simulation of process nets can be stopped at arbitrary states then process nets provide little information about goals; if a goal is not reached in a process net then this required property either does not hold true, or it holds true and a state marking a corresponding condition is reached in any continuation of the stopped process net. Our termination criterion of the simulation algorithm guarantees that such a situation cannot occur [De02]. This termination criterion takes the difference between transitions that belong to the control algorithm model and transitions that belong to the environment model into account; the environment is allowed to stop performing actions eventually whereas the control should always continue to proceed unless it runs into a deadlock situation.

More generally, goals can require that a set of causally independent conditions hold eventually. In our example, one could require that eventually all beverages are warm. Provided that the system behaves in a fair manner, this requirement holds true. Notice that a sequential run might have no according reachable marking if the transition *brew* always occurs immediately before *dispense*. Hence, in general, goals addressing more than one condition can not be equivalently defined using sequential semantics.

A *fact* is a property that holds for all reachable markings of a system model (sometimes facts are called *invariant* or *stable* properties). Facts can be checked in runs by analyzing all reached states. This is simple for sequential runs because the markings reached during an occurrence sequence are explicitly represented. Certain facts can be checked more efficiently using causal runs, without enumerating all markings reached during the run. This claim will be demonstrated using our example:

The process net of Figure 8 can be used to show that $\{ready, counter\}$ is not a fact implemented in the model. For this purpose, it is not necessary to play the token game. It rather suffices to find conditions labeled by *ready* and *counter*, respectively, that are not ordered, i.e., not connected by a directed path. In fact, the condition *counter* in the post-set of the first *insert* event and the second condition *ready* are not ordered. Every set of unordered conditions is a subset of a maximal set of unordered conditions. Every such maximal set corresponds to a reachable state of the process net [BF88]. Every state of the process net corresponds to a reachable marking of the system net by Lemma 1. Hence, the two unordered conditions prove that there is a reachable marking of the system net that marks both places. So, if this fact belongs to the specification, we have to rule out this process net.

6 Extensions, Experiences, Conclusions

We have presented an approach for the synthesis of formal models of dynamic systems. In particular, we considered controlled automation systems that consist of an uncontrolled plant and a control software. The main purpose of the model is to specify the behavior of the controlled system, which implies requirements for the controller software. The approach considers the generation of initial system models (the plant) and of formal specifications of the requirements for the controlled system, which are implemented in the model step by step. The approach is based on the assumption that the user knows the desired runs of the system but tends to make errors when formalizing specifications for these runs. Thus, the core of the approach is a simulation based technique to generate runs from specifications and to visualize these runs for inspection by the user. We have argued that causal concurrent runs have important advantages in relation to sequential runs, which are considered usually, because they better capture relevant aspects of the behavior, allow a more efficient representation of behavior and allow for more efficient analysis methods with respect to system requirements.

This contribution only gave a rough survey on our approach. Missing details include theoretical results, algorithmic details and extensions of the model. In this section we like to mention some of the extensions because they turned out to be essential for industrial applications.

The first extension is that we employ *high-level Petri nets* rather than elementary place/transition Petri nets. High-level Petri nets are often called *Colored Petri Nets* (see [Je95]), because the tokens appear with different identities (colors). Actually, tokens can represent arbitrarily structured data which can be used to manage values given by sensors and control data. Moreover, if there are several identical components of a system then it suffices to provide only one model for these components. The high-level tokens then also capture information about the actual component the token belongs to [DK01].

The second extension refers to specific modeling primitives for *signals*. Communication by signals is typical for controlled automation systems. In particular, the communication between control and plant is usually purely based on signals. There are two types of signal arcs in our variant of Petri nets [JL02]. *Event signals* asymmetrically synchronize two transitions of a Petri net: The synchronized transition occurs only if it is enabled and receives the signal from the other transition, sent by its occurrence. *Condition signals* allow to restrict the enabledness of a transition to those states where certain places carry specific tokens. Both signal arcs can also be used to transfer data from one transition to another one.

The third extension concerns *modularity* [JL02]. Each model is constituted by a set of modules. For example, the control and the plant of a system are both modeled by a separate module. Modules only communicate via signals, that is, only signal arcs connect net elements of different modules.

The concepts presented in this paper are partly implemented in the *VIPtool* that was developed by our group, see <http://www.informatik.ku-eichstaett.de/projekte/vip/>. A first version of the tool was written in the programming language Python. Recently, we began an improved implementation based on Java. Main features of the tool are a graphical Petri net editor for high-level Petri nets with signals, a simulation engine that generates causal runs, a visualization module that presents runs in a nice and readable way and moreover depicts the relation between process net elements and system net elements, and analysis tools for process nets that allow to validate given system requirements. Moreover, the tool allows to calculate parameters concerning time and cost such as average throughput time or average sum of costs for a given set of runs. These features allow to integrate quantitative requirements to process model generation.

Finally, we like to report on experiences with applying the approach in an industrial case study. This case study was developed together with engineers from the company Audi AG in Ingolstadt, Germany, and refers to the development of the forthcoming Audi A6 limousine. For legal reasons, it is not possible to publish details of the models obtained within this project.

Cars are particular instances of the class of controlled automation systems. We do not consider the driver controlling the car but software for the management and control of the various sensor data and according necessary actions to be performed in modern vehicles.

The considered example deals with the control of the fuel gauge and the corresponding indicator for possible remaining kilometers before refueling is necessary. Surprisingly, this topic has significant complexity. One of the reasons is that the actual assumed amount of remaining gasoline in the fuel tank is sometimes measured, with several sensors, and sometimes calculated from previous assumptions about this amount and the measured subsequent consumption. Parameters

that have direct influence to the calculation include the state of ignition, the state of car movement and the duration of a stop (surprisingly, no sensor recognizes whether the gas cap was opened). Moreover, different sensors indicate whether the car's position is at an acclivity. Due to the complicated shape of the tank, only some of the sensors actually contribute to the measurement, dependent on the actual fill level. Finally, each single sensor can fail, in which case a plausibility test rules out this sensor's data from the fill level calculation. The calculation based on consumed gasoline is quite precise, but small errors within a given tolerance range can finally add up to severe differences between the actual fuel level and the assumed fuel level.

The previous paragraph should provide enough information to show that it needs a nontrivial algorithm to control this technical system. The example contains continuous and discrete aspects. We applied our approach to derive a Petri net model of the controlled system. The size of this model has the magnitude of about 100 Petri net elements, it has several modules, and we heavily used high-level tokens and signal arcs.

The general feedback from the users, engineers from Audi, indicates that they considered the approach very useful. We were able to solve most of the problems posed by the users and, perhaps more importantly, we proved that the documents provided by Audi contained much more ambiguities and errors than expected by the users.

The main lesson we have learnt from this case study is the following. The assumption that users start with a vague description of the plant and several requirements and look for the controlling algorithm is only partly justified in this application area. Instead, very precise knowledge about the plant is available. This information has to be transformed in our modeling language which sometimes causes problems and needs feedback, because of hidden assumptions. The semi-formal requirement specification hardly includes an enumeration of safety and liveness properties the controlled system has to satisfy. These requirements are implicitly given and often go without saying (for example, the tank should not become empty without prior warning of the driver). Instead, the model work was based on desired scenarios of the style "what happens if...". In our terminology, a set of runs in a semi-formal style was provided, formalized in our approach, and validated by the experts. These runs have interfaces to the model of the plant. Each model of the control algorithm that supports these runs will also support additional, different runs and shows that situations can arise for which no scenarios were provided. Our simulation approach identifies these situations and offers runs to the user that are possible due to the respective model, this way enforcing the users to complete the necessary requirements. Thereafter, the formalized requirements are validated.

In this application area, real-time aspects play an important role. For example, the behavior of the control algorithm depends on the duration of a car stop. Real-time aspects are not yet integral part of our modeling language and our approach but can only be handled indirectly. Currently, we are working on the integration of this aspect, without abandoning the concurrent causal semantics though. In other words, we capture the notion of local time for single components but do not assume the existence of a global time scale that would press all events of a run into a total order, i.e., into a single sequence.

We are going to continue the cooperation with Audi and model more car components that are relevant to control. We hope to better exploit concurrency and causality between events that belong to different components when larger models are considered, than it was possible by now.

References

- [BF88] Best, E., C. Fernandez C.: *Nonsequential Processes*, Springer-Verlag, Berlin Heidelberg New York (1988)
- [De99] Desel, J.: Validation of System Models Using Partially Ordered Runs. In: Szczerbicka, H. (ed.): *Modelling and Simulation: A Tool for the Next Millenium*, Proc. of the 13th European Simulation Multiconference ESM'99, Warschau, Juni 1999, 295–302, Society for Computer Simulation (1999)
- [De00a] Desel, J.: Validation of Process Models by Construction of Process Nets. In: van der Aalst, W., Desel, J., Oberweis, A. (eds.): *Business Process Management*, Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 110–128
- [De00b] Desel, J.: Simulation of Petri Net Processes. In: Kozák, Š, Huba, M. (eds.): *Proc. of the IFAC Conference on Control System Design*, Bratislava, June 2000, 14–25 (2000)
- [De00c] Desel, J.: Teaching System Modeling, Simulation and Validation. In: Joines, J.A., Barton, R.R., Kang, K., Fishwick, P.A. (eds.): *Proc. of the 2000 Winter Simulation Conference (WSC'00)*, Orlando, December 2000, 1669–1675 (2000)
- [DE00a] Desel, J., Erwin, T.: Modeling, Simulation and Analysis of Business Processes. In: van der Aalst, W., Desel, J., Oberweis, A. (eds.): *Business Process Management*, Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 129–141
- [DE00b] Desel, J., Erwin, T.: Hybrid Specifications: Looking at Workflows From a Run-Time Perspective. *International Journal of Computer System Science and Engineering*, Vol. 15 No. 5 (2000) 291–302
- [De02] Desel, J.: Model Validation – A Theoretical Issue? In: Esparza, J., Lakos, C. (eds.): *Application and Theory of Petri Nets 2002*, Lecture Notes in Computer Science, Vol. 2360. Springer-Verlag, Berlin Heidelberg New York (2002) 23–43
- [DK01] Desel, J., Kindler, E.: Petri Nets and Components – Extending the DAWN Approach. In: Moldt, D. (ed.): *Workshop on Modelling of Objects, Components and Agents*. Aarhus, Denmark, DAIMI PB-553 (2001) 21–36
- [DR98] Desel, J., Reisig, W.: Place/Transition Petri Nets. In: Reisig, W., Rozenberg, G. (eds.): *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, Vol. 1491. Springer-Verlag, Berlin Heidelberg New York (1998) 122–173
- [JL02] Juhás, G, Lorenz, R.: Modeling with Petri Modules. In: Caillurd, B., Darondeau, P., Lavagno, L., Xie, X. (eds.): *Synthesis and Control of Discrete Event Systems*. Kluwer Academic Press (2002) 125–138
- [GR83] Goltz, U., Reisig, W.: The Non-Sequential Behaviour of Petri Nets. *Information and Computation*, Vol. 57 (1983) 125–147
- [GT76] Genrich, H., Thieler-Mevissen, G.: The Calculus of Facts. *Mathematical Foundations of Computer Science*, Springer-Verlag, Berlin Heidelberg New York (1976) 588–595
- [Je95] Jensen, K.: *Coloured Petri Nets, Vol. 1: Basic Concepts*. 2nd edition, Springer-Verlag, Berlin Heidelberg New York (1995)