

Model Validation in Controller Design

Jörg Desel, Vesna Milijic, and Christian Neumair

Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt
Ostenstr. 28, 85072 Eichstätt, Germany
{joerg.desel, vesna.milijic, christian.neumair}@ku-eichstaett.de

Abstract. This work considers model construction 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 originally 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 construction 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 is based on particular Petri nets, called signal nets, as formal process models and partially ordered runs as their semantics. This contribution also reports on a case study from the automotive industry.

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 use a tailored variant of Petri nets together with a process net semantics.

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 particular, we consider embedded systems in cars. 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 construction are the appropriate formalization of the requirements (and their validation) and the correct generation of the model from the formal specifications.

Model construction 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 construction, formalization and validation as one important early phase in the process of system development.

This work will present an approach for model construction for controlled systems that employs different formalization / validation steps and a synthesis procedure to obtain the model from the specifications in a systematic way. It also presents a case study developed with the car manufacturing company Audi (see also [8]) and reports on experiences with applying this method.

The basis of our formal modelling language are signal nets [11, 14, 15], an extension of Petri nets. In order to adapt our modelling language to industrial relevance, features for modularity, interaction between modules and differentiation between controllable, observable and internal events had to be integrated. Extensions also concern a timing concept for representing real time aspects and real-valued sensor data employing concepts of High-Level nets. The approach is based on simulation and verification. By simulation we mean construction and inspection of partially ordered causal runs, represented again by signal nets.

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, our formal modelling language is presented. The causal simulation of our extension of signal nets, its advantages and some words about algorithmic aspects is the topic of section five. Section six illustrates applying this approach to the industrial case study, also providing net models and partially ordered runs. Finally, experiences from the case study are outlined in section seven.

The first sections of this paper are strongly based on [2] and [5], where more details can be found. Different aspects of the approach were also adapted to and presented in various different communities (see [1], [3], and see [4] 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 adapt 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.

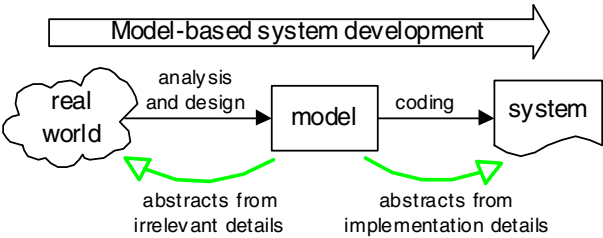


Fig. 1. Model based system development

In Figure 1, 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”. Figure 2 shows a more faithful representation of the situation.

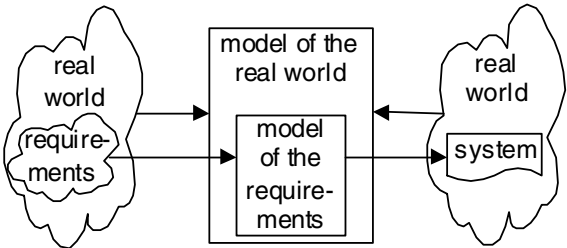


Fig. 2. Capturing the embedding in the real world

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.

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 im-

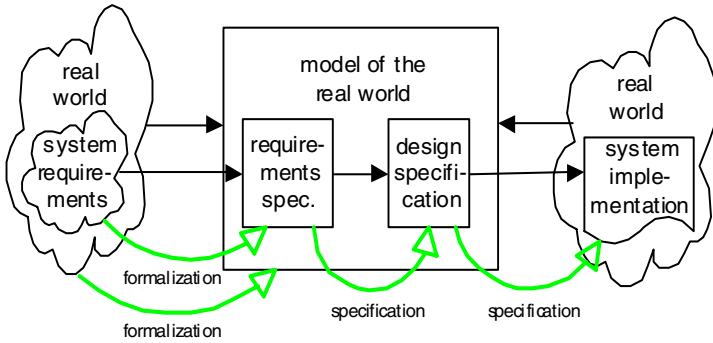


Fig. 3. Capturing requirements and design specifications

plementation, 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 Figure 4, 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.

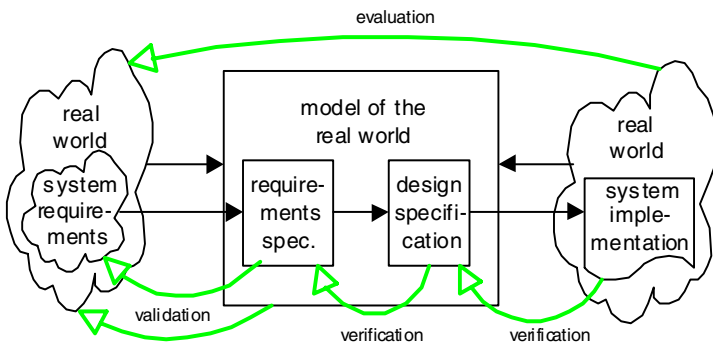


Fig. 4. The position of validation, verification and evaluation

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 valid 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 modelling 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 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 modelled. 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 our approach (see [15]). However, it will turn out that some verification means can also be used for validation purposes.

We first consider the problem of modelling 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 [3] 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 modelling 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.

We now 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 information 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. 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

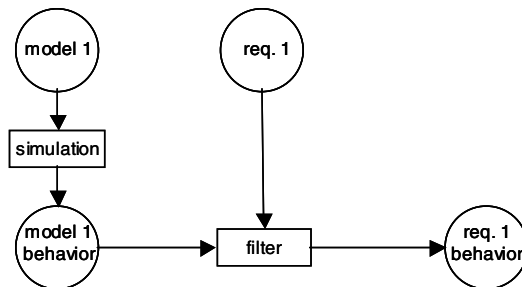


Fig. 5. A first step in requirements validation

properties (requiring that something eventually happens) and safety properties (requiring that something bad does not happen) are added in arbitrary order, then previous steps might have to be repeated.

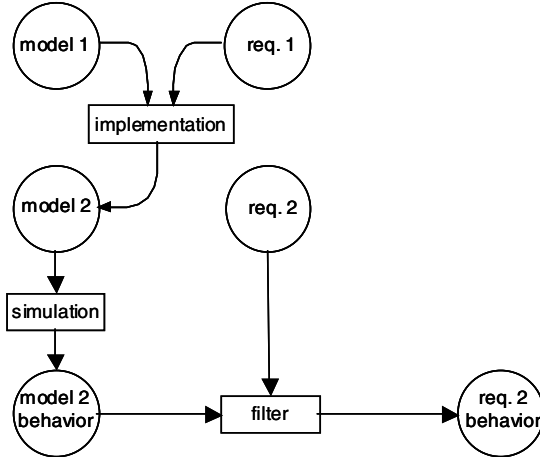


Fig. 6. A second step in requirements validation

4 The Modelling Approach

4.1 The Modelling Language

To model a system, we use signal nets [6, 7, 11, 15, 16] that are an extension of Petri nets. A signal net is, like a Petri net, a graph with two types of nodes. Our modelling language extends signal nets to principles like modularity, interaction between modules and differentiation between controllable, observable and internal actions. Also a timing concept for representing real time aspects and concepts of High-Level nets for representing real-valued sensor data are included.

Places. Each place may contain tokens from a place-specific defined set of token types, called domain. The same token can appear more than once in a place. To depict output-places, where tokens represent data to be read from outside, grey background color is used. We distinguish two kinds of places, namely low-level and high-level places.

- *Low-level places* are represented by a simple circle. The domain of low-level places contains a single token type: a black token. The number of black tokens represents the actual state of the place. If these places represent conditions then in any reachable state they contain at most one black token: one token means that the condition is fulfilled, no token means it is not. This will be the case in the examples given in section six.

- *High-level places* are represented by a double-framed circle. The domain of these places is an arbitrary nonempty set of token types. The state is symbolized by the number of tokens which belong to each token type. In the examples, high-level places will only contain one token and the domain will be always the set of real numbers. Hence, to simplify matters, the state of a high-level place will be symbolized by a real number.

Transitions, drawn as rectangles, represent actions. Actions that do not underly the control algorithm, called uncontrollable, like user driven actions or errors will be symbolized by a darker grey background. Transitions with a light grey background characterize actions that generate new values, for example for sensor data. These transitions and the white colored transitions are controllable.

Arcs. Nodes can be connected by different kinds of arcs.

- *Flow arcs* are black arcs that either connect a place with a transition or a transition with a place. They may have a time label. In the case they connect high-level places and transitions or vice versa, they are labelled by a variable.
- *Read arcs* are double-sided black arcs between transitions and places. If they connect high-level places and transitions, they are labelled by a variable and may also have a time label.
- *Write arcs* are double-sided grey arcs between transitions and high-level places that are labelled by two variables and possibly a time label.
- *Synchronization arcs* connect two transitions and are graphically represented by jagged arcs.

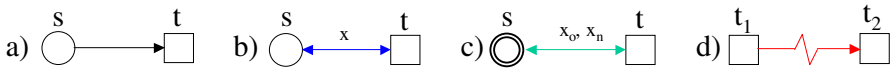


Fig. 7. a) Flow arc b) Read arc c) Write arc d) Synchronization arc

A state of a signal net is determined by its current distribution of tokens in the places, also called *marking*. We denote the initial distribution of tokens by *initial marking*. The *dynamic behavior* of a signal net is given by the firing of transitions. The surrounding arcs determine whether a transition may fire and how its firing changes the marking.

Transitions that are not connected to high-level places are called *low-level transitions*. If every low-level input place, connected to the low-level transition by a flow arc or a test arc, contains a token, this transition is enabled and may fire.

A transition which is connected by labelled arc with at least one high-level place is called *high-level transition*. Each variable of the labelled arc can be substituted by a value of the domain of the connected high-level place (so in our case by a real number). A high-level transition may have a *firing condition*,

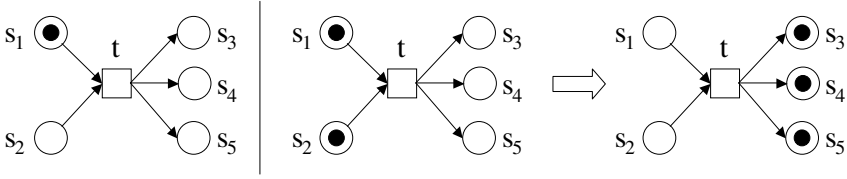


Fig. 8. On the left hand side you can see a net where transition t cannot fire because of the empty place s_2 . The right hand side shows a net where transition t is able to fire and the resulting state of the net after firing of t

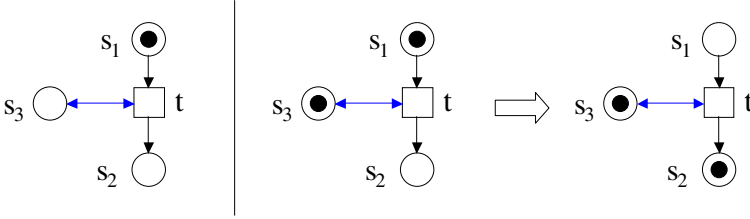


Fig. 9. On the left hand side, transition t cannot fire because of the empty place s_3 . On the right hand side, firing is possible

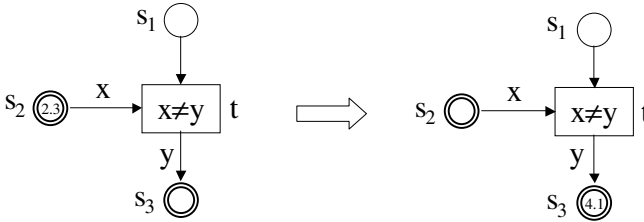


Fig. 10. This net contains two high-level places (s_2 and s_3). Place s_2 contains the token 2.3. Firing transition t substitutes x and y . While x has to be substituted by 2.3 (as the token of place s_2), variable y can be substituted by an arbitrary value but not the same as x . On the right side you can find the resulting net after firing with $x = 2.3$ and $y = 4.1$

i.e., a Boolean term that includes the variables from the labelled arcs connected with the transition. To fire a high-level transition, values for the variables at the surrounding arcs have to be substituted in the following way: The variable of an arc, resp. the first variable in the case of a write arc, leading from a high-level place to the transition is substituted by a token of this place. The substituted values must fulfill the firing condition of the transition. Moreover, each low-level input place, has to contain a token. The firing of a transition (high- or low-level) deletes a token from each low-level input place connected with a flow arc and produces a token in each low-level output place connected with a flow arc. Firing a high-level transition deletes the current value in each input high-level place and produces in every output high-level place the token

determined by the substitution of the arc variable. In addition, firing a high-level transition deletes the current value in every high-level place connected with the transition by a write arc, and the value that substitutes the second variable of the write arc is produced. Read arcs do not change the token of the place they are connected with. If an enabled transition is connected to another enabled

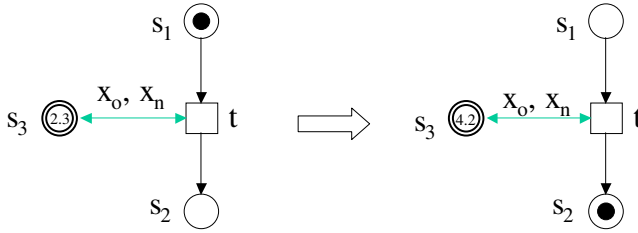


Fig. 11. This example shows a net with a write arc before and after firing of transition t with the substitution $x_a = 2.3$ and $x_n = 4.2$

transition by a synchronization arc, both transitions will fire at the same time (the first transition is synchronizing the second transition). A transition with an incoming synchronization arc will never fire without the synchronization signal, whereas a synchronizing transition can also fire alone. Our extension of signal nets also provides a concept of time. An enabled transition fires immediately after the time on the label of every arc with time label ingoing to the transition has expired after enabledness. A time label of an arc leading from a transition to a place causes that firing the transition produces a token in the place after the time of the time label has passed. The time label of a write arc between a place and a transition means that by firing the transition the old value is replaced by the new value after the time on the label has passed.

Controllable transitions that have only ingoing flow and write arcs without time label underly the progress assumption. This means that for each set of transitions which are pairwise in conflict, one will eventually fire. Two transitions, which are both enabled, are in conflict if after firing one transition the other one is no more enabled.

4.2 Causal Semantics

We now concentrate on process models, i.e., on specifications of runs of a system. 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 our extension of signal nets can be defined in several ways. We will discuss sequential runs, given by occurrence sequences and causal runs, given by process nets.

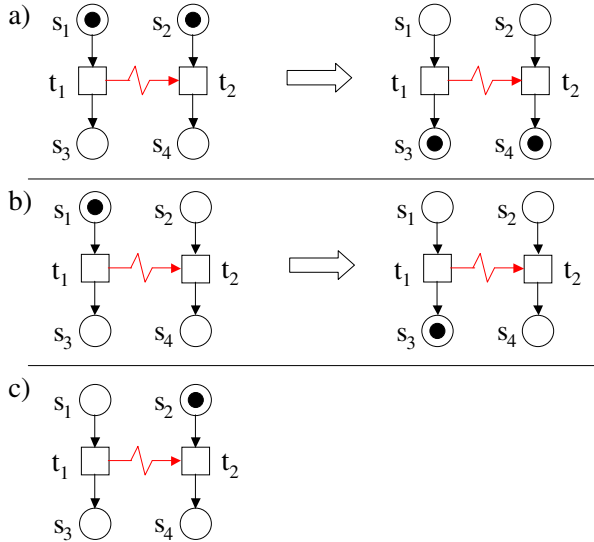


Fig. 12. a) In this net, transitions t_1 and t_2 are enabled to fire. Firing transition t_1 synchronizes transition t_2 , which then also fires. The net state before and after firing is presented. b) In the second case, only transition t_1 can fire and does not synchronize transition t_2 as the place s_2 contains no token. c) Transition t_1 cannot fire because of place s_1 contains no token. So transition t_2 will not fire because it is not synchronized by transition t_1

There are basically two different techniques to describe the behavior of our signal 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. 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. If transitions fire synchronously due to synchronization arcs, we combine the names of these transitions and regard them as one event. 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.

The second technique employs process nets representing causal runs. It constitutes the causal semantics of our extension of signal nets. Also process nets are extended signal nets. 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.

A causal run consists of a set of events (representing the firing of one or a set of synchronized transitions each), symbolizing 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. Combining events with their explicitly modelled pre- and post-conditions yields a causal run, formally represented by a process net.

Our extensions of signal nets make it necessary to take some further notes on causal runs concerning causal dependencies of events.

- A read arc either tests if a condition is fulfilled (low-level place) or reads some value (high-level place) but it does not change the token in the connected place. Hence, it is possible that more than one transition connected with the same place by read arcs can simultaneously access the place. Thus, read arcs do not influence the causal dependencies between events.
- In our modelling language, time aspects are modelled by time labels on some arcs, which either cause that transitions fire immediately after a certain time has passed or that tokens are produced after a certain time period. These time aspects have no effect on the formal system's behavior, i.e., they do not influence dependencies of events in a process net.
- Finally, remember that we consider the firing of synchronous transitions as one event.

In a process net, each token is produced by at most one transition occurrence, and it is consumed (remember that read arcs just test, but do not consume) by at most one transition occurrence. Hence, conditions of process nets are not branched w.r.t. flow and write arcs.

The immediate causal dependency of events is represented by the flow and write arcs of a process net. No two elements can be mutually causally dependent, in other words, the flow and write relation has no cycles. So the causal relation is a partial order that we call causal order. Two different events are causally ordered if and only if they are connected by a chain of directed flow or write arcs. Otherwise, they are not ordered but occur concurrently.

A condition of a process net represents the appearance of a token on a place of the original net and is therefore drawn as a copy of the place labelled by the name of the place. In case of high-level places, the copy also includes the current value of the high-level place.

As an event represents the occurrence of at least one transition, it is depicted as a copy of a transition of the original net. If the event represents the occurrence of a single transition, it is labelled by the name of the transition. If an event represents the occurrence of a set of synchronous transitions, it is labelled by all elements of the set of names of these transitions. Consequently, there are no synchronization arcs in a process net.

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. We assume that every event has at least one pre-condition and at least one post-condition. By this assumption, all minimal elements are conditions. 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.

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 signal net with our extensions and that there is a close connection between a process net representing a run and the extended signal net modelling 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 original 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.

Sequential and causal runs have strong relations. Sequences of event occurrences of a process net closely correspond to transition sequences of the system net. Therefore, roughly speaking, the set of occurrence sequences of an extended signal net coincides with the set of occurrence sequences of its process nets when only the labels of events of these latter sequences are considered.

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 modelled 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 extended signal 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, possible parallel behavior etc. are directly represented in process nets, but they are hidden in sequences of events. Causal runs provide full information about these causal dependencies.

The number of event occurrence sequences of a single run 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.

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. Taking the sequence of labels of events in occurrence sequences of process nets yields all occurrence sequences of the system net.

In previous publications, we have described the simulation algorithms [2, 5], i.e. an algorithm constructing runs. 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 [2]), adopting the well-known fact transitions [10] 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.

6 The Case Study

In the context of a new production run of cars, this case study with the car manufacturing company Audi was concerned with the control system of the fuel gauge of a car, which is surprisingly complex. The value of the fuel gauge is sometimes determined by various sensors and sometimes calculated by means of consumption and an earlier calculated fill level. Numerous parameters like ignition state, movement of the car, car position and engine on/off are relevant. Because of the special shape of the tank only the values of some sensors – depending on the current fill level – account for the calculation of the fuel gauge. If some sensor fails, a plausibility test avoids that the values of this sensor are used for the calculation. Though the consumption based calculation is rather exact, the summation of minor measurement errors below a certain threshold may lead to a significant difference between real and calculated fuel gauge. Already these problems indicate that complex algorithms are necessary for the control of this technical system which includes continuous and discrete elements.

Starting point of the case study was an informal, mainly textual document where functionality of the fuel gauge control system was described. The following tasks arose in the project:

1. Adaption of appropriate techniques for modelling and model synthesis,
2. development of models of control for the fuel measurement,

3. simulation and validation of the models,
4. feasibility study for similar tasks but with larger scale,
5. specification of the requirements of software tools to be developed.

6.1 Procedure

When starting the project, we expected to receive a basic model of the uncontrolled system and a set of informal and formal requirements [12, 13]. However, actually the input of Audi corporation was a completely different one: informal descriptions of scenarios were given, that had to be realized by the algorithm to be modelled, including some implicit requirements. The basic model of the plant was partly explained by corresponding automotive components (e.g. the functionality of the four tank sensors) and partly assumed as well-known (e.g. possible states of ignition or of car movement). Some of these implicit assumptions could easily become completed, others made it necessary to query at Audi corporation. Interpreting the description of the scenarios, ambiguities were detected. Even by Audi corporation, some ambiguities could not be cleared up instantly. In general, the (not surprising) assumption was affirmed, that modelling strongly depends on feedback of experts and cannot solely be done by the given documents. So, in the procedure, the first steps of modelling and especially of a repeated validation turned out to be of high importance [2, 5].

In the following, we present our intended procedure. In our case study, the later phases of model based generation of test cases are presented only exemplarily.

Procedure

Steps one to six refer only to single modules.

1. Extraction and validation of the model of the plant.
2. Extraction, formulation and validation of the requirements.
3. Modelling of runs from the scenarios comprising the model of the plant.
4. Generation of the complete system by folding the runs. This complete system includes the given runs but may also have some other (desired or undesired) ones.
5. Elimination of undesired runs by implementation of the requirements found in step two. It has to be guaranteed that the given runs of step three are still possible.
6. Validation of the single modules and verification of the requirements.
7. Integration of the modules by composition at certain restrictive interfaces.
8. Verification of the complete system.
9. Application as reference model.
10. Analysis for efficient generation of relevant test data.
11. Application as test reference by simulation, in parallel to the real control.

6.2 Model of Calculation of Fuel Gauge while *Ignition off*

In the following three subsections we present the case study. Due to lack of space not every detail can be explained. Each subsection contains a short description of the given specification of the corresponding model. Then the model, including both plant and control is presented. A description of its behavior is visualized by different process nets.

In each model, the plant is represented by dark grey colored transitions and the places connected with these transitions. For simplicity, those parts of the plant, which are not necessary for the control algorithm in the current model are omitted. If a complete model is desired, the models can be connected by identifying common parts of the plant.

The calculation of the fuel gauge while *ignition off* consists of two measurement phases. The first measurement phase starts by turning the ignition off and returns the mean value *MW old*. Turning the ignition on starts the second measurement phase and returns the mean value *MW new*. If the difference between the two mean values exceeds 4 liters, refuelling is recognized. Then the *displayed fuel* is recalculated by adding this difference to the *old value*. The following requirements must be fulfilled for calculating the fuel gauge:

1. If the period of time while the ignition is off is too short to finish measurement phase one, there will be no new calculation of the fuel gauge.
2. The result of measurement phase 1 outlasts if ignition is turned on for a short time.

In the left part of the model the behavior of the ignition is presented by its physical (1, 2) and internal (1', 2') transitions. When the ignition is off for 6 seconds, measurement phase one starts. 4 seconds later the first phase ends by firing transition 5 what produces a value in the place *MW old*. Turning the ignition on initiates the second measurement phase (transition 6), if measurement phase one has already been finished (indicated by a token in the place *synchronization*). If the ignition stays turned on for 0.4 seconds, *MW new* is calculated. Ignition on and a difference between *MW old* and *MW new* greater than 4 liters are preconditions for a change of the fuel gauge value: the transition with firing condition $|x - y| \geq 4L$ fires and the difference is added to the former value in place *old value*. If the difference is less than 4 liters, no refuelling is recognized: the transition with firing condition $|x - y| < 4L$ fires and the marking of the place *displayed fuel* remains unchanged.

At this point the calculation is declared finished (token in place *calculation completed*), to enable – when the ignition is turned off next time – a new calculation of the fuel gauge value.

The following three pictures show relevant process nets of the model: first, a complete calculation and change of the fuel gauge value is done (Fig. 14). In the second process, calculation is aborted because of turning on ignition before ending of first measurement phase (Fig. 15). The last process shows an interrupted calculation: after the first measurement phase, when the second is started by turning ignition on, the driver could turn off again ignition before the second

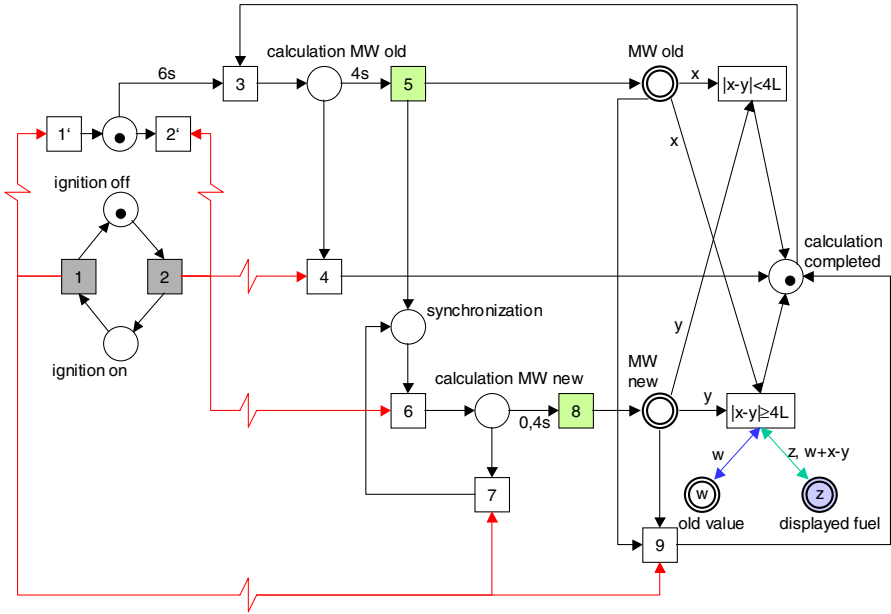


Fig. 13. Signal net model of the fuel gauge while the ignition is off

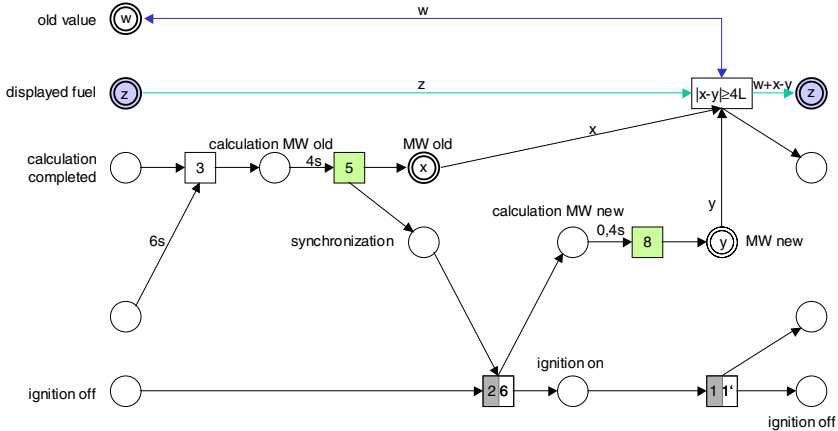


Fig. 14. Process net of a complete calculation of the fuel gauge value

measurement phase can happen. In this case, the result of measurement phase one will be remembered until ignition is turned on and the second measurement phase is executed (Fig. 16).

After some iteration steps, we came up with the following formalization of the requirement, which holds for all simulated runs.

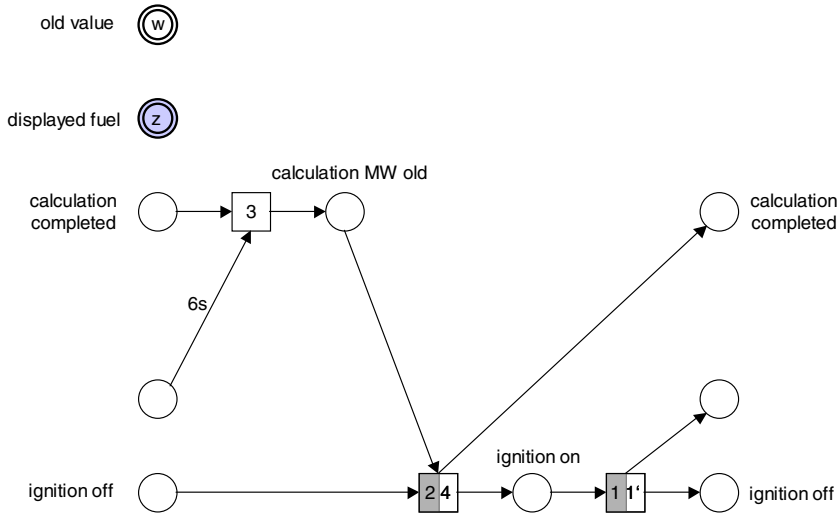


Fig. 15. Process net of an abort of the calculation during the first measurement phase

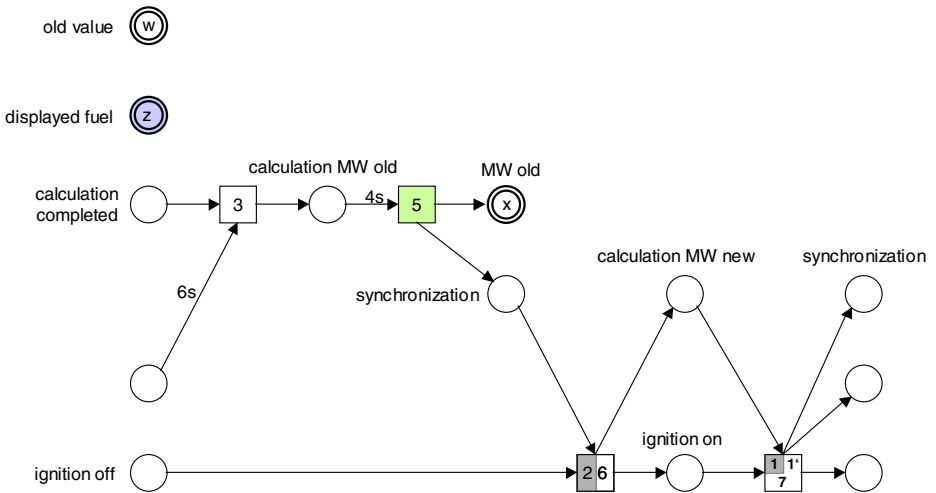


Fig. 16. Process net of an interrupted calculation during the second measurement phase

If exactly the places $MW\ old$, $ignition\ off$, $displayed\ fuel$, $old\ value$ and $synchronization$ are marked and then transitions 2 and 1 fire sequentially within 0.4 seconds, then the marking in place $MW\ old$ stays unchanged.

The verification of this formalized requirement will be exemplarily shown.

A sufficient precondition is that always exactly one of the three places *calculation MW old*, *MW old* and *calculation completed* is marked. This property can be formally proved by so called place invariants. As in our requirement place

MW old is marked, the other two places are not marked. So transitions 3 and 4 cannot fire while *MW old* remains marked.

The only enabled, not synchronized transition is transition 2 which fires by assumption. As transition 6 is enabled it will be synchronized by transition 2. Therefore place *calculation MW new* is marked. By assumption, transition 1 fires within 0.4 seconds after transition 2. Before transition 8 can fire, the place *calculation MW new* is marked for 0.4 seconds. So transition 7 is enabled and synchronized when transition 1 fires. Thus the token in *calculation MW new* is consumed and a token in place *synchronization* is produced, recovering the initial marking.

6.3 Model of Calculation of Fuel Gauge while *Ignition on* and *Vehicle Stops*

The calculation of the fuel gauge while *ignition on* and *vehicle stops* also requires two measurement phases. Measurement phase one starts if the vehicle stops for 8 seconds while the ignition is on. This phase lasts 4 seconds and returns the mean value *MW old*. Subsequently, measurement phase 2 returns repeatedly a new mean value *MW new*. If refuelling is recognized once (difference between *MW old* and *MW new* greater than 4 liters), the fuel level is repeatedly recalculated by the *MW new* values by adding the difference of the mean values *MW new* and *MW old* to the *old value*. Among others, the following (informal) requirements must be fulfilled:

1. Recognizing refuelling once, the time period for detecting the mean values of *MW new* is shortened from 2 seconds to 0.4 seconds.
2. The calculation of the mean value *MW new* lasts until the vehicle moves again or the ignition is turned off.

The left part of the model shows the physical and internal states and the possible changes of states of the ignition and of the vehicle (*vehicle stops*, *vehicle moves*). To start measurement phase 1 (transition 5), the ignition has to be turned off for 8 seconds and the vehicle must not move. After 4 seconds the calculation of *MW old* is completed (transition 6). Afterwards the calculation of the mean value of *MW new* starts (transition 7) and is completed after 0.4 seconds (transition 8), if the ignition stays turned on during this time period. Directly after the calculation of *MW new* (time label 0 seconds), it is checked if the difference between *MW old* and *MW new* is greater than or equal to 4 liters. If this is the case, refuelling is recognized (transition with firing condition $|x - y| \geq 4L$ fires). If this difference is less than 4 liters, no refuelling is recognized (transition with firing condition $|x - y| < 4L$ fires), and then every two seconds a new mean value *MW new* is calculated (transition 9) and the above procedure is repeated again. Once refuelling has been recognized a new mean value *MW new* is yet calculated every 0.4 seconds (transition 10) and then immediately (time label 0 seconds) the value of the fuel gauge is actualized (transition 11). This happens by adding the difference of the mean values to the *old value* (transition 11) until

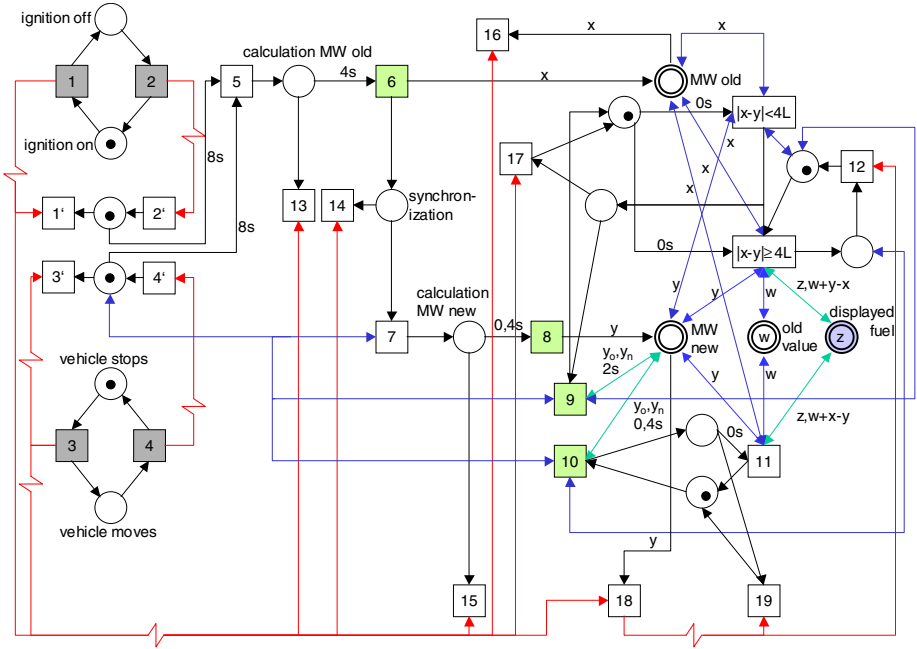


Fig. 17. Signal net model of fuel gauge while *ignition on* and *vehicle stops*

either the ignition is turned off (transition 1) or the vehicle moves (transition 3). In these cases the recalculation of the value of the fuel gauge is stopped and the initial marking in the right part of the model is restored again. This allows a new calculation of the fuel level if the car stops the next time for 8 seconds while the ignition is on.

As an example, the following figures show scenarios given by partial ordered runs of the above model.

We provide a valid formalization of requirement 1 and outline its verification:

After firing of the transition with firing condition $|x - y| \geq 4L$, transition 9 must not fire but transition 10. It fires alternately to transition 11 until either transition 1 or transition 3 fires.

When the calculation of the fuel gauge starts, the place between transition 12 and the transition with the firing condition $|x - y| \geq 4L$ is initially marked. This marking is a precondition for firing transition 9 (calculation of the *MW new* values every two seconds). Analogously, a precondition for firing transition 10 is that the place between the transition with the firing condition $|x - y| \geq 4L$ and transition 12 is marked. As always exactly one of both places contains one token (this property can be formally proved by so called place invariants), it follows: if transition 9 can fire, then transition 10 is not enabled and otherwise. So after firing the transition with the firing condition $|x - y| \geq 4L$, only transition 10 and 11 can alternately fire until transition 12 which is synchronized by transition 1 and 3, fires.

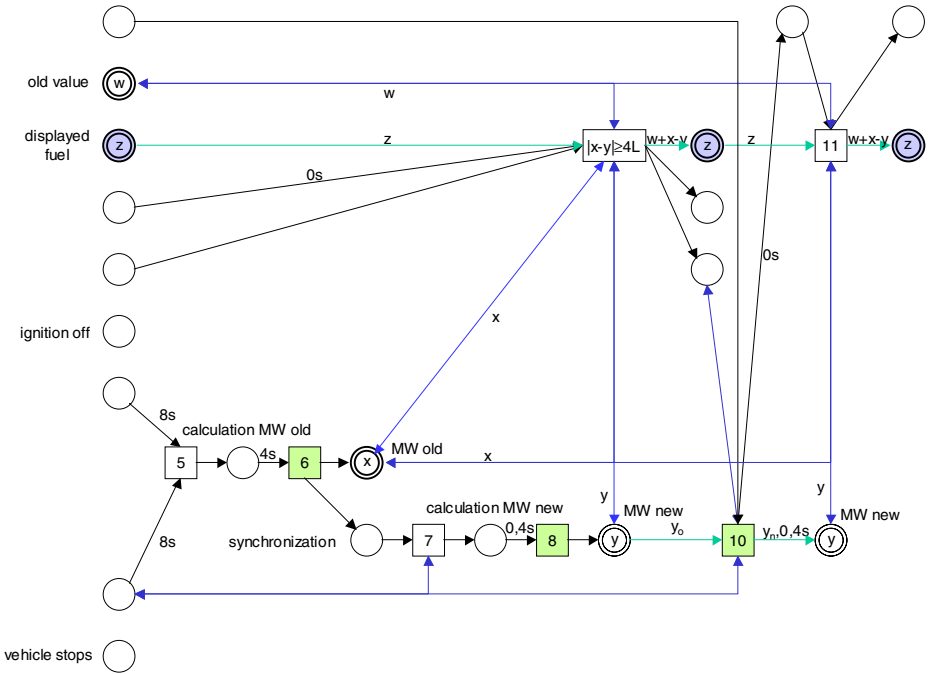


Fig. 19. Process net of the calculation of a new fuel gauge value, because refuelling is recognized

The signal-net model consists of four parts: in the left part, the possible physical states (*sensor ok*, *sensor broken*, *sensor shorted*) and the possible changes of states of the sensor are modelled. Only if the sensor is *ok*, the actual sensor value x (given by transition 1) is used as ADC-value. Otherwise if *sensor broken* is given, a maximal value $max > So$ is used and in case of *sensor shorted* a minimal value $0 < Su$. In the right lower part the ignition is modelled. The middle and right part show the algorithm for the error treatment of the sensor. In the middle part the treatment of the ADC-value is done which either can be normal (place *ADC-value: ok*), to small (place *ADC-value: too small*), or to large (place *ADC-value: too large*) depending on the measured sensor value in the place *sensor value*. For this purpose the transitions are labelled by the corresponding firing conditions. In the right part, depending on the classification of the ADC-value, the internal detection of a sensor error and the kind of the sensor error is described. This detection only occurs if ignition is on (place *ignition on*). As an example the following figures show scenarios given by partial ordered runs of the above model.

Finally, requirement 3 is formalized and verified:

If for at least 20 seconds the places sensor broken and ignition on are simultaneously marked, the place break will be marked after these 20 seconds.

Obviously requirement 1 has to be integrated as a basic condition for an error treatment of a sensor. The following considerations are necessary for the

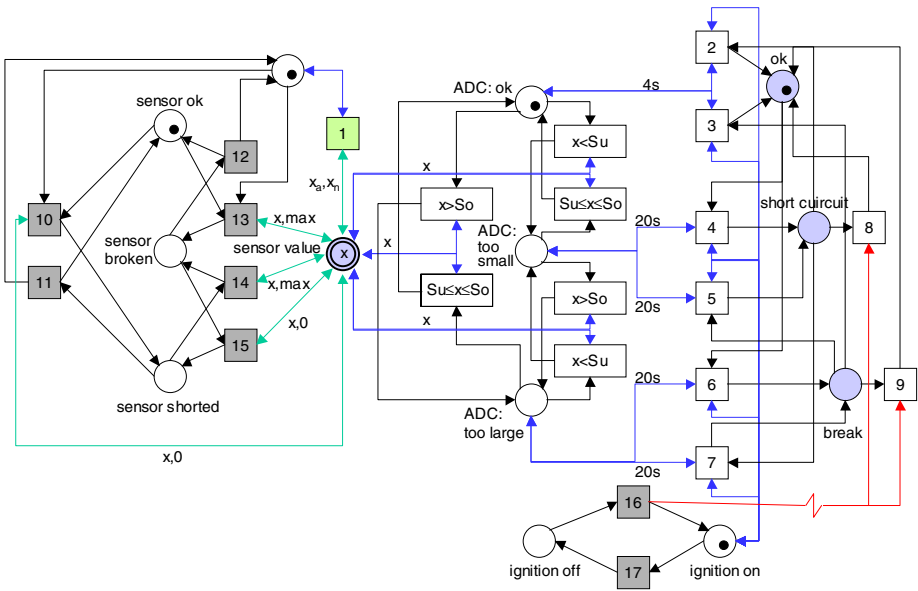


Fig. 20. Signal net model of error treatment for one sensor

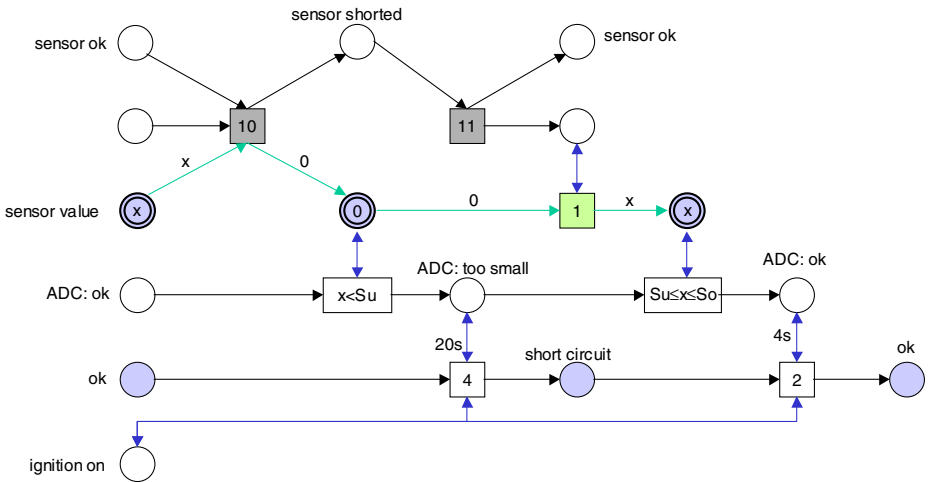


Fig. 21. Process net describing that a short circuit is recognized first, then the sensor is ok again

verification of requirement 3: in both, the middle and the right part of the model always exactly one place is marked with one token (so the corresponding sets of places are place invariants). Furthermore, in both parts of the model always exactly one transition is enabled what implies that there is no conflict between

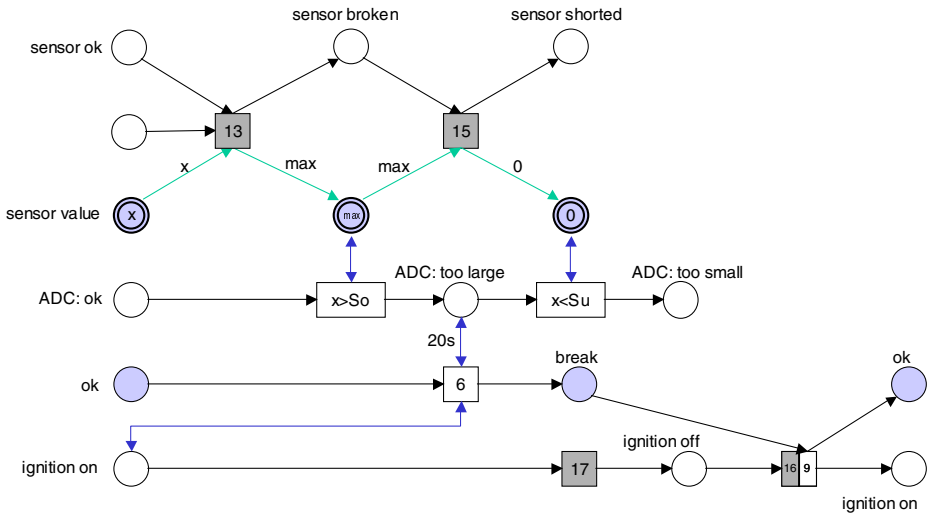


Fig. 22. Process net describing that break is recognized first, a following short circuit is not recognized as the ignition is turned off/on

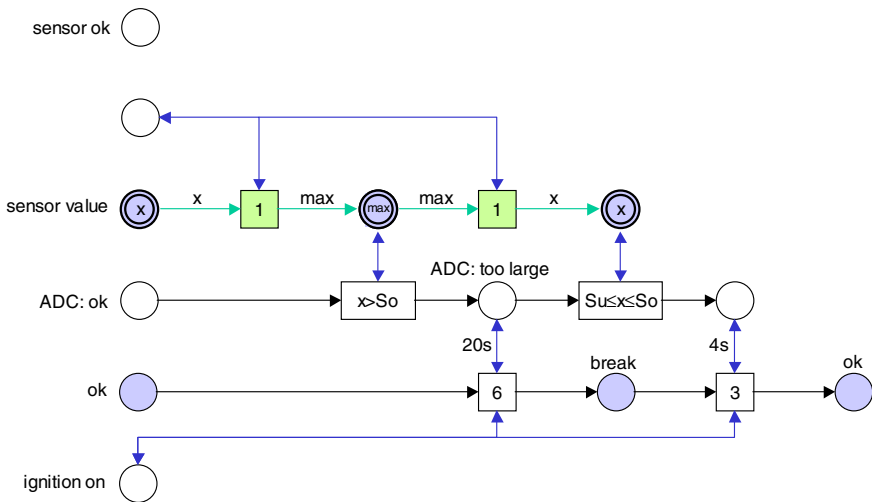


Fig. 23. Process net describing that break is recognized first, then the sensor is ok again

these transitions. To mark the place *sensor broken* either transition 13 or 14 has to fire. In particular, the place *sensor value* is then marked with the value *max*. As the place *sensor broken* stays marked for 20 seconds (as postulated in the requirement), during this period of time no transition in the left part fires. So for at least 20 seconds, the place *sensor value* is marked by the value *max*. Because of our previous considerations, exactly one transition of the middle part of the

model with the firing condition $x > So$ can fire. This transition fires because of the progress assumption as it is in no conflict with any other transition of the other parts of the model (there are no flow arcs between the different parts of the model). Afterwards the place *ADC-value too large* is marked and the token stays there for at least 20 seconds (as for this period of time there is no transition enabled which could change this marking). Now the place *break* of the right part of the model is marked and so either no transition of this part is enabled or exactly one of transition 6 or 7 is enabled (depending on the former ADC-value). Because of the progress assumption, one of these transitions fires after 20 seconds and marks the place *break*.

6.5 Generation of Test Cases

We illustrate, exemplarily for the model of error treatment of a sensor, the generation of test cases resp. test vectors:

The values of the input components of a test vector are supplied by the grey resp. light grey colored transitions and correspond to the marking of certain places: these places represent sensor data resp. interfaces to components which deliver sensor data or control parameters. In this model (*sensor ok*, *ignition on*, *max*) is an input vector. The first component represents the state of the sensor delivered by transitions 10 to 15, the second component the state of the ignition (delivered by transitions 16 and 17) and the third component the sensor value (delivered by transition 1). The value of the output components corresponds to the marking of the grey colored places. Suitable values for the test vectors resp. sequences of test vectors can be determined by the model based range of values. If necessary, additional information has to be requested if the range of values does not matter for the algorithm (e.g. the maximal fill level that can be displayed by a sensor; it is determined by the dimension of the tank and the scale of the sensor and is important for plausibility test but not for other algorithms). It is not surprising that for the determination of local test vectors from a model there cannot be extracted more than it was explicitly put in. The advantages of model based determination of test vectors rest in more global aspects. The consideration of dependency resp. independency of input and output values that can be won by analyzing the model reduces the number of necessary combinations of test values extensively. For example, it can be derived by the model that *ignition on* and an incorrect *sensor value* are necessary to detect a sensor error. Therefore not all input combinations of (*sensor ok*, *sensor broken*, *sensor shorted*) and (*ignition on*, *ignition off*) have to be tested as for *ignition off* the first parameter plays no role.

This approach is based on the assumption that the actual realized control algorithm has the same dependencies as the modelled algorithm. Both should be equivalent concerning the specification, but this equivalence can be only supported by tests. So, in the best case, the model based generation of test vectors can give hints for relevant test data. But it cannot be excluded that such errors in the control, which could be detected by other test data, remain hidden. Such test data can only be delivered by further information about the realized

algorithm, which cannot be derived from the specification or the model which is only based on the specification.

7 Experiences and Conclusions

After a general discussion of model validation, we have presented an approach for the systematic construction of formal models of embedded 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 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.

We further presented our extension of signal nets as the modelling language to be used and discussed as an industrial case study the fuel gauge control of a car. We illustrated the extended signal net models and some of the causal runs and gave an idea of how to verify given requirements.

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 modelling 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 modelling 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.

To draw a conclusion, the feedback from the users, engineers from Audi corporation, indicates that they considered our 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 corporation contained much more ambiguities and errors than expected by the users.

The concepts presented in this paper are partly implemented in the VIPtool [9] that was developed by our group, see <http://www.informatik.ku-eichstaett.de/projekte/vip>. Main features of the tool are a graphical net editor, 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.

References

1. 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, Society for Computer Simulation, 295–302, 1999.
2. Desel, J.: Validation of Process Models by Construction of Process Nets. In van der Aalst, W., Desel, J., Oberweis A. (Eds.): *Business Process Management*, LNCS 1806. Springer-Verlag, 110–128, 2000.
3. Desel, J.: Simulation of Petri Net Processes. In Kozk, ., Huba, M. (Eds.): *Proc. of the IFAC Conference on Control System Design*, Bratislava, 14–25, 2000.
4. 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, 1669–1675, 2000.
5. Desel, J.: Model Validation - A Theoretical Issue? In Esparza, J., Lakos, Ch. (Eds.): *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets 2002*, ICATPN 2002, LNCS 2360. Springer-Verlag, 23–43, 2002.
6. Desel J., Juhás G., Lorenz R.: Process Semantics and Process Equivalence of NCEM. In *Proc 7. Workshop Algorithmen und Werkzeuge für Petrinetze AWPN 2000*, Fachberichte Informatik, Universität Koblenz - Landau, 7–12, 2000.
7. Desel J., Juhás G., Lorenz R.: Input/Output Equivalence of Petri Modules. In *Proc. of the 6th Biennial World Conference on Integrated Design and Process Technology IDPT 2002*, Pasadena, California, 2002.
8. Desel J., Juhás G., Lorenz R., Milijic V., Neumair Ch., Schieber, R.: Modellierung von Steuerungssystemen mit Signal-Petrinetzen – eine Fallstudie aus der Automobilindustrie. In Schnieder, E. (Ed.): *8. Fachtagung Entwurf komplexer Automatisierungssysteme 2003*, Proceedings of EKA 2003, Braunschweig, 273–297, 2003.
9. Desel J., Juhás G., Lorenz R., Neumair Ch.: Modelling and Validation with Vip-Tool In *Proc. of the International Conference on Business Process Management*, LNCS 3080, Springer-Verlag, 2003.
10. Genrich, H., Thieler-Mevissen, G.: The Calculus of Facts. *Mathematical Foundations of Computer Science*, Springer-Verlag, 588–595, 1976.
11. Hanisch H.-M., Lüder A.: A Signal Extension for Petri nets and its Use in Controller Design. *Fundamenta Informaticae*, 41(4), 415–431, 2000.
12. Hanisch H.-M., Lüder A., Rausch M.: Controller Synthesis for Net Condition/Event Systems with a Solution for Incomplete State Observation. *European Journal of Control*, (3), 280–291, 1997.

13. Hanisch H.-M., Thieme J., Lüder A.: Towards a Synthesis Method for Distributed Safety controllers Based on Net Condition/Event Systems. *Journal of Intelligent Manufacturing*, (5), 357–368, 1997.
14. Juhás G., Lorenz R.: Modelling with Petri Modules. In Caillaud, B., Darondeau, Ph., Lavagno, L., Xie, X. (Eds.) *Synthesis and Control of Discrete Event Systems*, 125–138, Kluwer, 2002.
15. Juhás G., Lorenz R., Neumair Ch.: Modelling and Control with Modules of Signal Nets. In Desel J., Reisig W., and Rozenberg G. (Eds.): *Lectures on Concurrency and Petri Nets*, LNCS, Springer, 2004 (In this volume).
16. Sreenivas R. S., Krogh B. H.: Petri Net Based Models for Condition/Event Systems. In *Proceedings of 1991 American Control Conference*, vol. 3, 2899–2904, Boston, MA, 1991.