Gold, Robert

Working Paper

Petri Nets in Software Engineering

Arbeitsberichte - Working Papers, No. 5

Provided in Cooperation with:
Technische Hochschule Ingolstadt (THI)

Suggested Citation: Gold, Robert (2004) : Petri Nets in Software Engineering, Arbeitsberichte - Working Papers, No. 5, Fachhochschule Ingolstadt - University of Applied Sciences, Ingolstadt, https://nbn-resolving.de/urn:nbn:de:bvb:573-203

This Version is available at:
https://hdl.handle.net/10419/202557

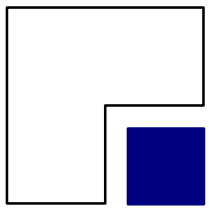
Fachhochschule
Ingolstadt
University of
Applied Sciences

Arbeitsberichte
Working Papers

Kompetenz schafft Zukunft
Creating competence for the future

Petri Nets in Software Engineering

von Prof. Dr. Robert Gold



Fachhochschule
Ingolstadt

University of
Applied Sciences

Arbeitsberichte
Working Papers

Petri Nets in Software Engineering

von Prof. Dr. Robert Gold

Heft Nr. 5 aus der Reihe
"Arbeitsberichte - Working Papers"
ISSN 1612-6483

Ingolstadt, im Juni 2004

Abstract

In this paper we investigate the use of Petri nets in software engineering extending the classical software development process with simulation and mathematical analysis based on place/transition nets. The advantage is that requirements can be validated earlier and fault detection and correction is less expensive. We show how to construct nets from basic patterns and demonstrate this for an application in automotive electronics, the cruise control with distance warning. The resulting nets can be simulated and analysed using Petri net tools and embedded into an object-oriented framework, where transitions are triggered by messages.

0 Introduction

Almost every process model for software development is build around the phases requirements analysis, preliminary and detailed design, implementation, integration and test. The *V-model* (Figure 1), an extension of the well-known waterfall model, depicts the development phases in the form of the letter V with the verification activities module test, integration test and system test between them [Bal98], [Bal00], [Car02], [Ger97], [Jal97].

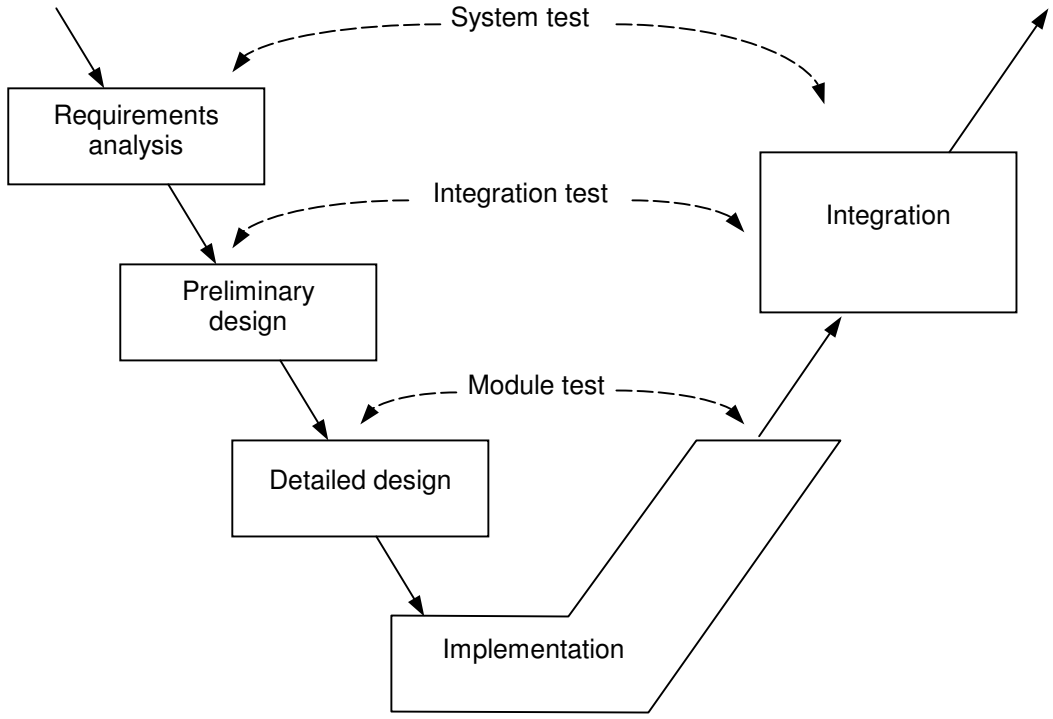


Fig. 1: V-model

One inherent problem of such process models is that, since the first executable software emerges not before the end of the integration, the requirements cannot be validated earlier. This means that faults introduced in the requirements analysis or the design phase due to misunderstandings or forgotten requirements remain in the software until the final validation and therefore they are expensive to detect and correct [Bal00], [Boe81], [Jal97]. Rapid prototyping and prototyping models improve this situation, but they are also very expensive since prototypes should not be used in the further development process.

We need means to validate earlier without unnecessary or additional costs. The solution is simulation using an executable model of the requirements together with mathematical proof techniques. Of course building the model will create considerable additional effort, but it is returned by lower fault correction costs and also by the possibility of code generation. An additional benefit of code generation is to avoid faults which are introduced by hand-coding.

The executable model of the requirements complements the textual or semi-formal specification and thus forms a new kind of specification paradigm, which combines the best of formal and informal specification of software.

Simulation requires an executable and hence a formal model of the software. It should be a mathematical model in order to allow mathematical proof techniques. On the other side the model should be feasible in practice for software engineers and not only for specialists. A graphical visualisation of the model is equally important to enhance the understanding of the software requirements.

The rest of the paper is structured as follows. In Chapter 1 we introduce the V^2-model which extends the V-model with simulation, validation and code generation. Basic definitions about place/transition nets are given in Chapter 2. In Chapter 3 we show how to construct nets from patterns and simulate the resulting nets. In the following Chapter 4 some mathematical methods for analysis are summarised. How to integrate the Petri net model into an object-oriented framework and to generate code is shown in Chapter 5.

1 The V^2-Model

The output of each activity in the V-model (Figure 1) is one or more development products. The requirements analysis produces the software requirements, the design phases a set of design documents, the implementation the source code and the integration the software executable on the target hardware. Other documents are test and quality management documents, e.g. test specifications, and project and configuration management documents. If we restrict the set of products to requirements, source code and executable software, we get Figure 2.

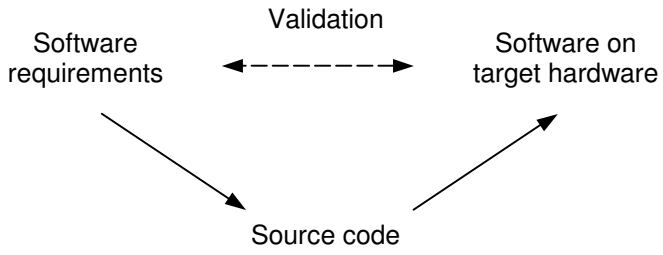


Fig. 2: V-model of development products

Validation shows whether the software satisfies the requirements. When modelling and simulation is used, the requirements can be validated before the source code is created and the source code can be derived from the model. The software is no longer verified against the requirements, but against the model (Figure 3).

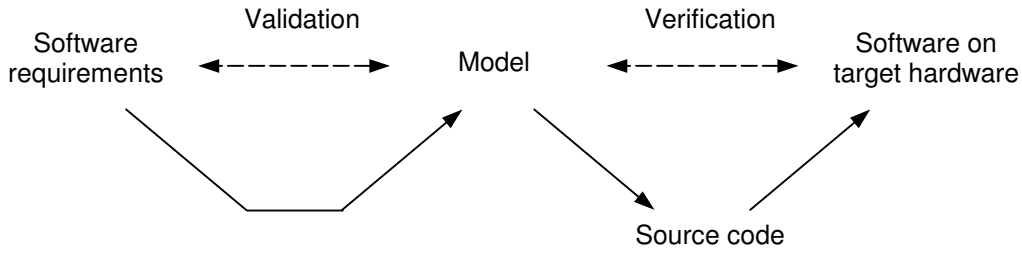


Fig. 3: V-model with simulation

The source code can be generated from the model, if a formal mathematical notation is used for the model. Since the model abstracts from implementation and hardware details, the code generation has to fill them in. If this can be done automatically and if the code generator itself is verified, verification of the software is unnecessary. But in many cases the generated code has to be optimised manually and therefore verification can not be omitted, e.g. if for the integration in a real-time operating system additional timing information is needed, which cannot be captured in the model, but it is crucial for fulfilling the required functionality.

This process is described by the *V^2-model* (Figure 4). Dot-dash lines mark activities which can be carried out automatically unless hand-coding or hand-optimisation is necessary [Gol02].

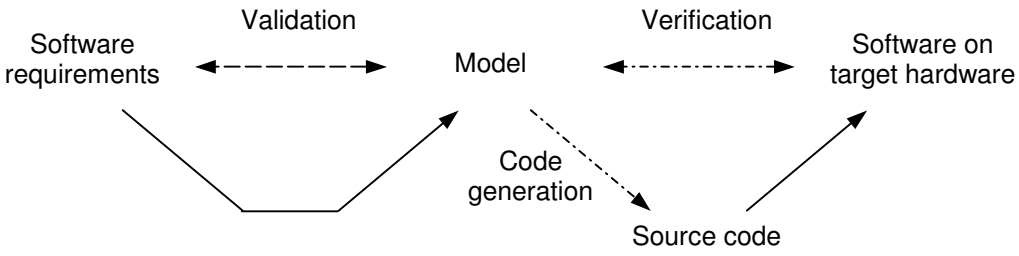


Fig. 4: V^2-Model

2 Place/Transition Nets

Petri nets are a well-known formal model which combine a rich mathematical theory with a useful graphical notation. Amongst the many different types of Petri nets place/transition nets form a simple but in many cases practically sufficient net class [Bau96], [RR98].

Definition: A fourtuple $N = (S, T, F, M_0)$ is called *place/transition net* or *net* for short, if

S is a finite set of *places*,

T is a finite set of *transitions* with $S \cup T \neq \{\}$ and $S \cap T = \{\}$,

$F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*,

$M_0: S \rightarrow \mathbf{N}_0$ is the *initial marking* function.

Places model local states of the system, transitions the actions. State changes are modelled by the flow relation. The flow relation connects places with transitions and transitions with places, but not elements of the same type. The initial state of the system is represented by the initial marking (Figure 5). For better understanding places and transitions will sometimes be labelled uniquely by strings. An example net is shown in Figure 6.

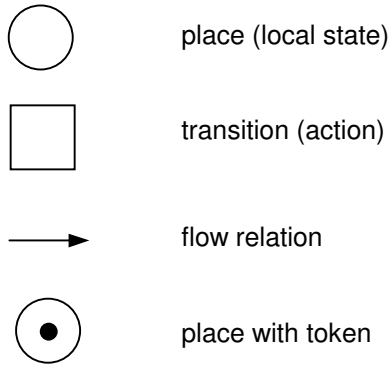


Fig. 5: Graphical notation of nets

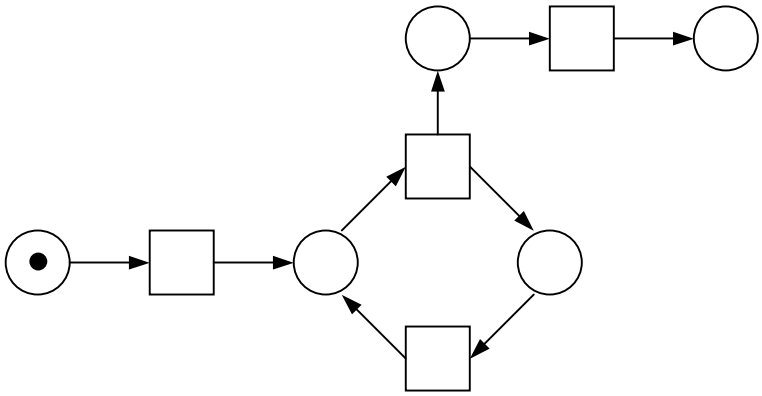


Fig. 6: Example net

States of the system are represented by markings of the net and depicted by black dots (tokens) in the places. The dynamic behaviour can be described by the flow of tokens initiated by the firing of transitions.

Definition: Let $N = (S, T, F, M_0)$ be a net. A function $M: S \rightarrow \mathbf{N}_0$ is called a *marking* of N. A transition $t \in T$ is *enabled* at a marking M if for all $s \in S$ with $(s, t) \in F$: M(s) ≥ 1. The set of such places is called *preset* of the transition. The set of places $s \in S$ with $(t, s) \in F$ is called *postset* of the transition. An enabled transition may *occur*, yielding the *follower marking* M' with $M'(s) = M(s) - 1$ for all places s in the preset but not in the postset of the transition, $M'(s) = M(s) + 1$ for all places s in the postset but not in the preset of the transition and $M'(s) = M(s)$ for all other places s. This is denoted by $M\ [t\rangle\ M'$. The set of *reachable markings* $[M_0\rangle$ is the smallest set of markings of N such that $M_0 \in [M_0\rangle$ and if $M_1 \in [M_0\rangle$ and $M_1\ [t\rangle\ M_2$ for $t \in T$ then $M_2 \in [M_0\rangle$.

An example for transition occurrences and token flow is shown in Figure 7.

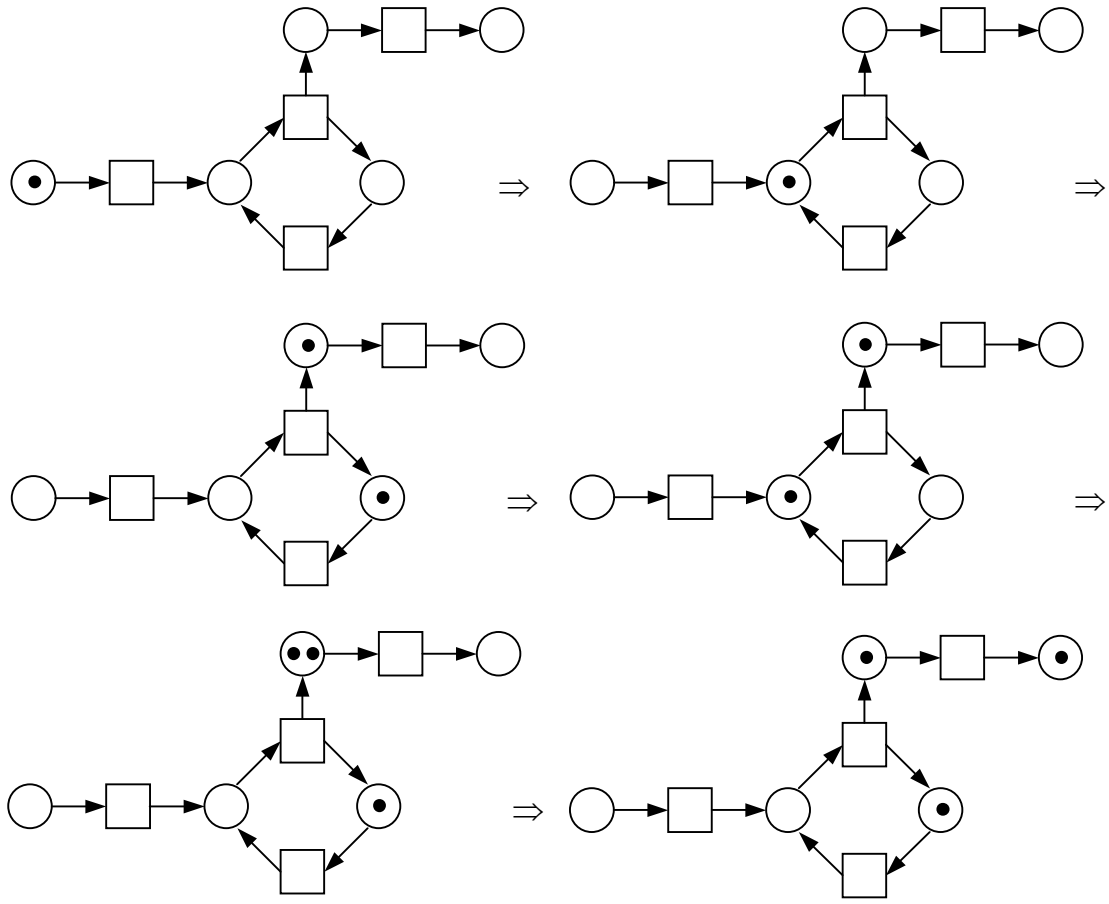


Fig. 7: Example for transition occurrences

For a more detailed introduction to place/transition nets see e.g. [Bau96], [GV03], [RR98].

3 Construction of Nets from Patterns

In the analysis phase we have to construct a net from informal requirements. In our approach we identify patterns in the requirements and compose the net of them by place fusion. This composition method has been thoroughly studied in literature e.g. in [BG94], [Gol95].

We find in nets the typical basic patterns action, branch, merge and synchronisation of concurrent subsystems (Figures 8, 9, 10).

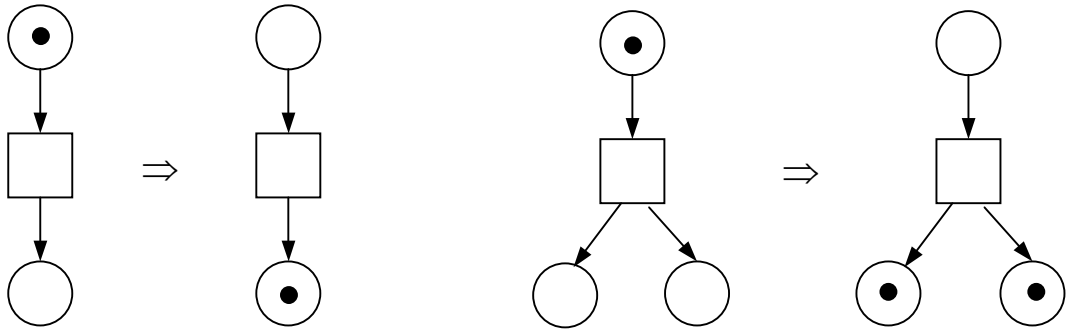


Fig. 8: Basic patterns action and branch of concurrent subsystems

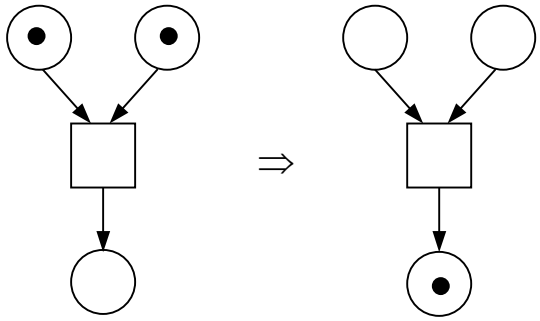


Fig. 9: Basic pattern merge of concurrent subsystems

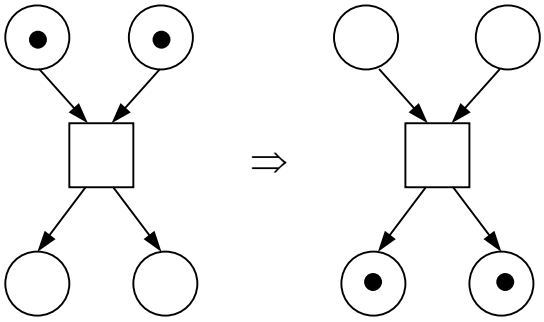


Fig. 10: Basic pattern synchronisation of concurrent subsystems

In general we have *n*-*m*-patterns that is transitions with *n* places in their presets and *m* places in their postsets. By fusion of places the patterns causality, concurrency and conflict can be composed from the basic action pattern (Figure 11).

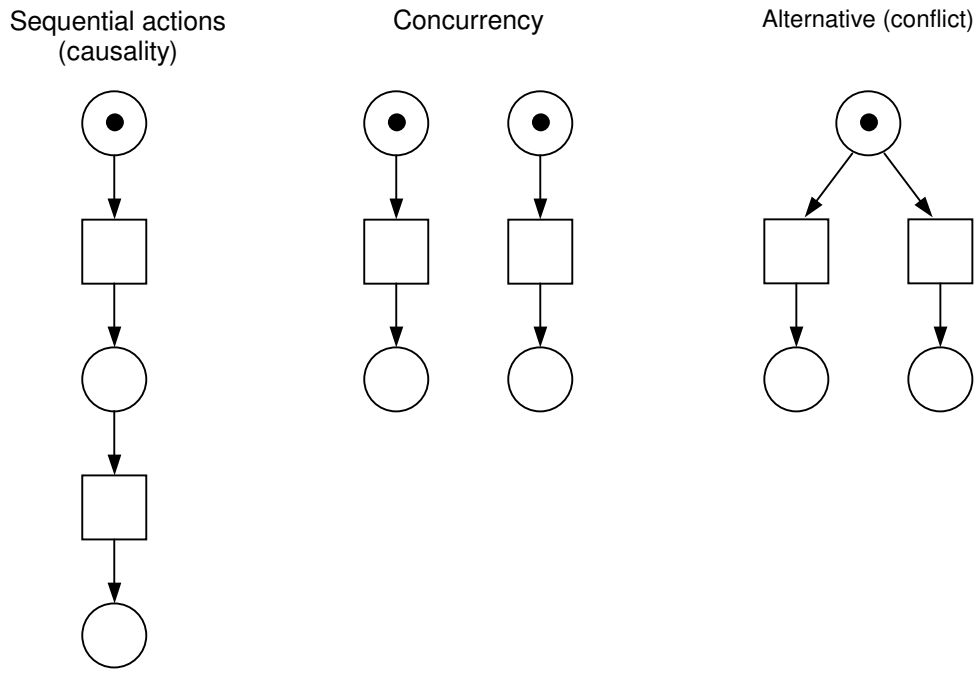


Fig. 11: Composed patterns causality, concurrency and conflict

Example: We will demonstrate the construction of nets for the cruise control with distance warning in automotive electronics (Figure 12).

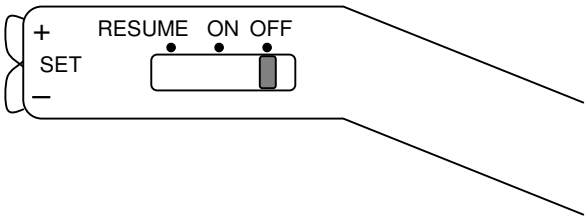


Fig. 12: User interface of the cruise control with distance warning

Let us consider a cruise control with the following functionality. After the cruise control is turned on (slider to ON) the actual velocity of the vehicle can be stored with the SET-button (button to +) and will be held constantly on this control value. Using the SET-button again (button to + or –) the value of the velocity is incremented or decremented by 2 km/h. If the driver uses the brake of the vehicle the control of the velocity is suspended. It can be resumed (slider to RESUME and back to ON). In suspended state the actual velocity is compared with the stored control velocity and a buzzer is activated for one second if the control velocity is exceeded. The cruise control is turned off by pushing the slider to OFF.

If we ignore the turning off of the cruise control we can distinguish the actions t_1 (turn on), t_2 (store velocity), t_3 (increment velocity), t_4 (decrement velocity), t_5 (brake), t_6 (control velocity exceeded) and t_7 (resume) with causalities $t_1 \rightarrow t_2 \rightarrow t_5 \rightarrow t_7$ and conflicts $t_3 \leftrightarrow t_4 \leftrightarrow t_5$, $t_6 \leftrightarrow t_7$ and get the (ccd-) net as shown in the left part of Figure 13.

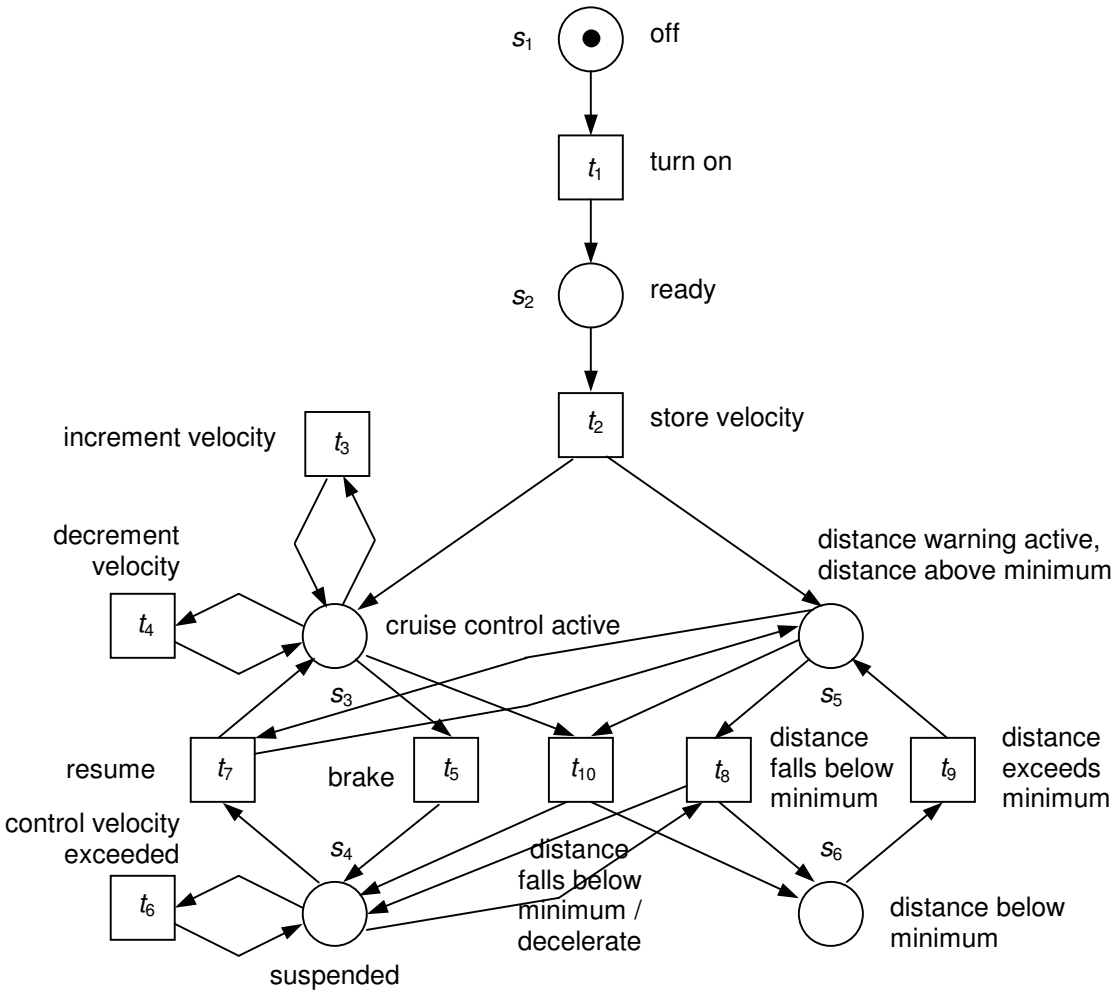


Fig. 13: Net for the cruise control with distance warning

Now we extend the cruise control by a distance warning functionality. Together with the first storing of the velocity the concurrent measurement of the distance to the vehicle driving in front is activated. The measured distance is compared to a computed minimal distance. If the measured distance is lower than the computed and the cruise control is activated, the control is suspended, the vehicle is decelerated and the driver is informed by a warning lamp.

We get the actions t_8 (measured distance falls below minimal distance) and t_9 (measured distance exceeds minimal distance) with causality $t_8 \rightarrow t_9$. The action t_2 will be extended to a branch in concurrent subsystems. A third action t_{10} models the synchronisation of the concurrent subsystems since the cruise control and the distance measurement are concurrent but not independent (Figure 13).

Since action t_8 does not suspend the cruise control, we have to make sure that this action only happens if the control is already suspended. On the other hand we have to avoid that the control is resumed if the distance is below the minimum, in other words is has to be above the minimum (Figure 13).

A net resulting from this composition process can be simulated using Petri net tools in order to validate the functionality of the model.

A list of Petri net tools can be found on the home page of the Petri Net World http://www.daimi.au.dk/PetriNets or in [Wik97].

Example: We specify simulation runs of the ccd-net:

Run 1 (activate the ccd): turn on – store velocity

Run 2 (change the control velocity): turn on – store velocity – increment velocity – decrement velocity

Run 3 (suspend the ccd because of slower vehicle in front, accelerate and resume the ccd): turn on – store velocity – distance falls below minimum / decelerate – distance exceeds minimum – control velocity exceeded – resume

Run 4 (suspend the ccd by braking, accelerate and resume the ccd): turn on – store velocity – brake – control velocity exceeded – resume

Run 5 (suspend the ccd by braking, change the velocity of the front vehicle and resume the ccd): turn on – store velocity – brake – distance falls below minimum – distance exceeds minimum – resume

4 Analysis with Reachability Graphs, Linear Invariants and Model Checking

Equally important for early validation as simulation is mathematical analysis. We show three of the most accepted methods: deadlock analysis with reachability graphs, linear invariant analysis and model checking.

Definition: A net $N = (S, T, F, M_0)$ is called *deadlock-free*, if for all reachable markings there is an enabled transition, i.e.

$$\forall\, M \in [M_0\rangle : \exists\, t \in T, M' \in [M_0\rangle : M\,[t\rangle\, M'$$

Example: The graph of reachable markings (reachability graph) of the ccd-net is shown in Figure 14 where markings are written as row vectors.

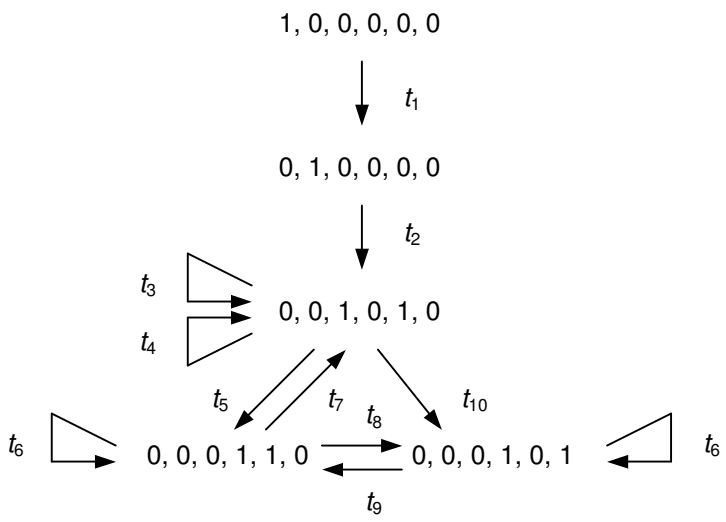


Fig. 14: Reachability graph of the ccd-net

The net is deadlock-free, since each node in the reachability graph has at least one outgoing edge.

The definition of linear invariants is based on the description of the flow relation of nets by matrices.

Definition: The *incidence-matrix* of a net $N = (S, T, F, M_0)$ with $S = \{ s_1, ..., s_n \}$ and $T = \{ t_1, ..., t_m \}$ is the matrix $C = (c_{ij})_{i = 1, ..., n, j = 1, ..., m}$ defined by

$c_{ij} = 1$, if s_i in the postset but not in the preset of the transition t_j

$c_{ij} = -1$, if s_i in the preset but not in the postset of the transition t_j

$c_{ij} = 0$, otherwise

If markings are interpreted as column vectors with n components and $M_1 [t_j\rangle M_2$, we have

$$M_2 = M_1 + C \cdot e_j$$

where e_j is the j-th unit vector.

Example: The ccd-net has the following incidence matrix:

$$C = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{pmatrix}$$

Let $M_1 = (0, 0, 1, 0, 1, 0)^T$ and $M_2 = (0, 0, 0, 1, 0, 1)^T$. We have $M_1 [t_{10}\rangle M_2$, $C \cdot e_{10} = (0, 0, -1, 1, -1, 1)^T$ (10^{th} column of C) and $M_2 = M_1 + C \cdot e_{10}$.

Definition: A *place invariant* $y: S \rightarrow \mathbf{Z}$ of a net $N = (S, T, F, M_0)$ is a solution of $C^T \cdot y = 0$. The set of place invariants of a net form therefore a vector space. A place invariant y is called *non-negative* if $y(s) \geq 0$ for all $s \in S$.

Let y be a place invariant. Then

$$\forall\ M \in [M_0\rangle : y^T \cdot M = y^T \cdot M_0$$

(token conservation law).

Example: The ccd-net has the following set of place invariants:

$$\{ y = (\lambda + \mu, \lambda + \mu, \lambda, \lambda, \mu, \mu)^T \mid \lambda, \mu \in \mathbf{Z} \}$$

The non-negative place invariant $(1, 1, 1, 1, 0, 0)^T$ means that the sum of tokens in the places s_1, s_2, s_3, s_4 is constantly 1 in each reachable marking, the cruise control is either off, ready, active or suspended. The same holds for the invariant $(1, 1, 0, 0, 1, 1)^T$ and the distance measurement.

In the rest of the chapter we present the linear time logic and model checking following [GV03].

Absence of deadlocks as defined above is a property of the net that can be proved by checking all reachable markings. Other properties depend on executions of the net and therefore need a language that takes into account that the truth of a formula can change over time. Such properties can be classified into safety and liveness properties [AS87]. Safety means that nothing undesired happens, liveness on the other hand states that a required property is satisfied by all executions of the net.

We consider a temporal logic called LTL [Pnu81] which is a restriction of the very general temporal logic CTL*.

Definition: A *LTL (linear time logic) formula* is either

- an atomic proposition, i.e. a condition on the number of tokens in a place or
- composed of LTL formulae: $\neg f$, $f_1 \wedge f_2$, $\circ f$, $[\, f_1 \cup f_2 \,]$ where f, f_1, f_2 are LTL formulae.

The operator $\Rightarrow$ is derived as usual, i.e. $f_1 \Rightarrow f_2$ means $\neg(f_1 \wedge \neg f_2)$.

The semantics of LTL formulae is defined using the reachability graph of the net, where we add for each node without successor an edge from the node to itself. A formula f holds for a net N with initial marking M_0 and a fixed infinite path $(M_0, M_1, M_2, \ldots)$ in the reachability graph, denoted by $< N, M_0 > \models f$:

$< N, M_0 > \models p$ $\quad\Leftrightarrow$ the atomic proposition p holds in the marking M_0

$< N, M_0 > \models \neg f$ $\quad\Leftrightarrow$ not $< N, M_0 > \models f$

$< N, M_0 > \models f_1 \wedge f_2$ $\quad\Leftrightarrow$ $< N, M_0 > \models f_1$ and $< N, M_0 > \models f_2$

$< N, M_0 > \models \circ f \quad \Leftrightarrow \; < N, M_1 > \models f$

$< N, M_0 > \models [\, f_1 \cup f_2 \,] \;\Leftrightarrow$ it exists $i \geq 0$ such that for all $0 \leq j < i$:

$< N, M_j > \models f_1$ and $< N, M_i > \models f_2$

The temporal operators *sometimes* and *always*, denoted by F and G respectively, are defined by

$$F\, f = [\, \text{true} \cup f \,]$$

$$G\, f = \neg F \neg f = \neg [\, \text{true} \cup \neg f \,]$$

where f is a LTL formula. These operators state that f holds sometimes on the path, i.e. at least at one state, and always on the path, i.e. at all states of the path, respectively.

Example: We consider the net for mutual exclusion as shown in Figure 15. One interesting liveness property of the mutual exclusion algorithm is “Each process that requests the critical section will obtain it”. This means that for an arbitrary execution always holds that if the place wait1 holds a token the place cs1 will be marked sometimes later and the same for process 2. This is expressed by the LTL formula f

$$G\, [\, \text{wait1} = 1 \Rightarrow F\, (\text{cs1} = 1) \,] \;\wedge\; G\, [\, \text{wait2} = 1 \Rightarrow F\, (\text{cs2} = 1) \,]$$

where $s = 1$ means that the place s contains one token. We have to prove

$$< N, M_0 > \models f$$

In the following we present an automata-theoretic approach for verification of LTL formulae. The idea is that a property can be characterised not only by a formula but also by an automaton, the so-called *Büchi automaton*, that accepts the set of behaviours which satisfies the property. By intersection with the set of infinite behaviours of the net represented by the reachability graph we find the set of behaviours of the net that satisfies the property. Another possibility is to construct the Büchi automaton that accepts the set of behaviours which satisfies the negation of the property. If the intersection with the set of behaviours of

the net is non-empty, we proved that the negation is true and therefore the property does not hold.

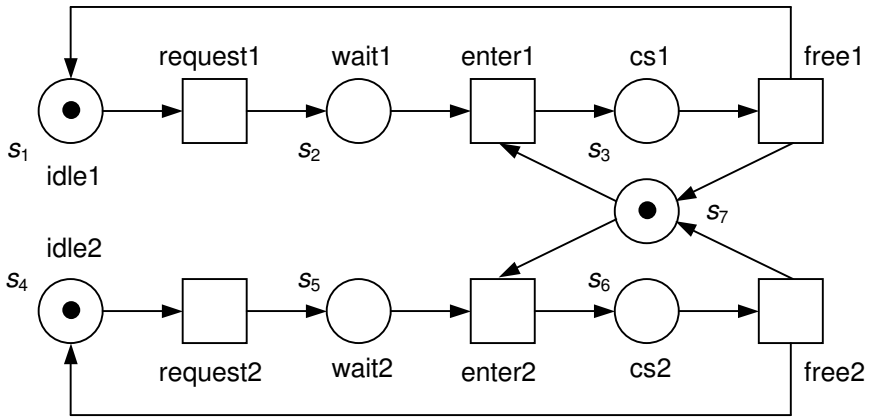


Fig. 15: Mutual exclusion net

The intersection of the sets of behaviours is constructed by the *synchronised product* of the reachability graph and the Büchi automaton. In this product the states are pairs (n, x) where n is a state in the reachability graph and x is a state of the Büchi automaton. There is a transition from (n, x) to (m, y) if and only if

- it exists a transition from n to m in the reachability graph,
- it exists a transition from x to y in the Büchi automaton labelled by a condition c and
- c is true in n.

The initial states of the product are the pairs of states where the components are initial states in the reachabilty graph and the Büchi automaton respectively. A state is accepting state in the product, if the corresponding state in the Büchi automaton is accepting.

If there is an infinite path in the synchronised product of the reachabilty graph and the Büchi automaton for the negation of the property, that encounter infinitely often an accepting state, there is an execution of the net where the liveness property does not hold.

Example: The negation of the property of the mutual exclusion net in the example above is

$$F\,[\,wait1 = 1 \wedge G\,(cs1 < 1)\,] \;\vee\; F\,[\,wait2 = 1 \wedge G\,(cs2 < 1)\,]$$

and can be characterised by a Büchi automaton (Figure 16).

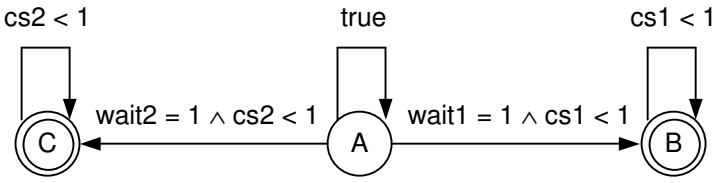


Fig. 16: Büchi automaton for the negation of the property of the mutual exclusion net

The initial state is A, the accepting states are B and C. The transitions are labelled by conditions on markings.

The reachability graph is shown in Figure 17 where M_0 = (1, 0, 0, 1, 0, 0, 1), M_1 = (0, 1, 0, 1, 0, 0, 1), M_2 = (1, 0, 0, 0, 1, 0, 1), M_3 = (0, 0, 1, 1, 0, 0, 0), M_4 = (0, 1, 0, 0, 1, 0, 1), M_5 = (1, 0, 0, 0, 0, 1, 0), M_6 = (0, 0, 1, 0, 1, 0, 0), M_7 = (0, 1, 0, 0, 0, 1, 0).

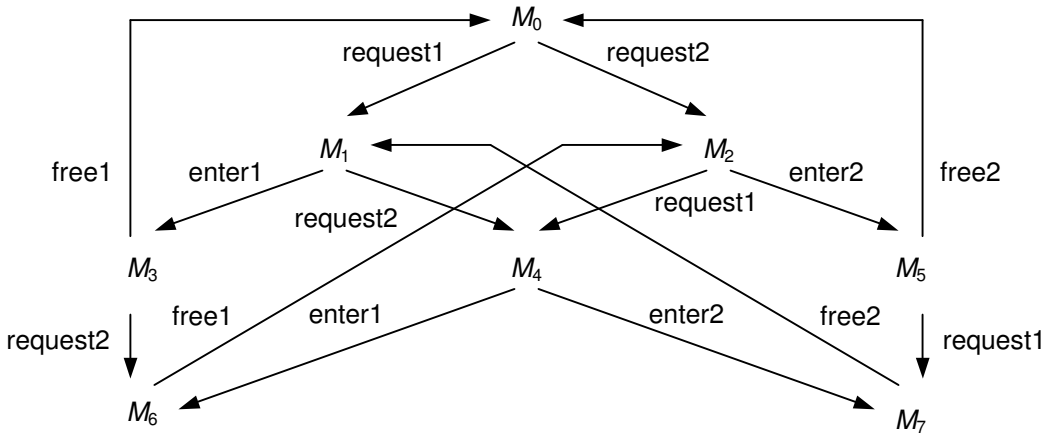


Fig. 17: Reachability graph of the mutual exclusion net

The synchronised product (Figure 18) has $8 \cdot 3 = 24$ states with initial state (M_0, A).

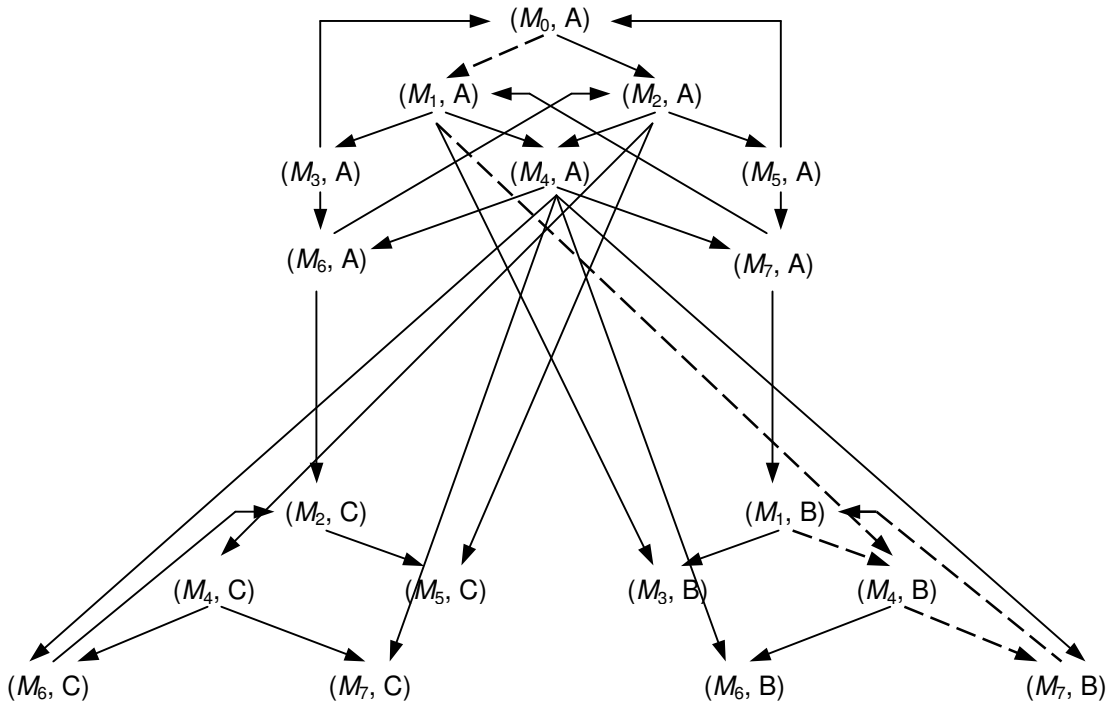


Fig. 18: Synchronised product for the negation of the property of the mutual exclusion net

Parts of the graph not reachable from the initial marking have been omitted in Figure 18. In the graph an infinite path, that encounter infinitely often an accepting state, exists, e.g. the path drawn in dashed lines. On this path process 1 waits for the critical section without entering it. That means that the property does not hold in the net.

Proving a liveness property therefore means detection of cycles in graphs which can be done in time linear in the size of the graph. In our example we build up the graph entirely and then looked for cycles. The number of nodes in the synchronised product is in the worst case the product of the numbers of nodes of the reachability graph and the Büchi automaton and therefore can be extremely high, possibly too high to store in memory. So-called *on-the-fly* methods compute the cycles without building up the graph entirely. Details can be found in the literature e.g. in [GV03].

5 Object-Oriented Design and Code Generation

The Petri net model is only one part of the overall, in most cases object-oriented design. We introduce a class for each net with a private array attribute for the marking (Figure 19). Transitions become methods that are triggered by messages and may themselves call methods of other classes (Figure 20).

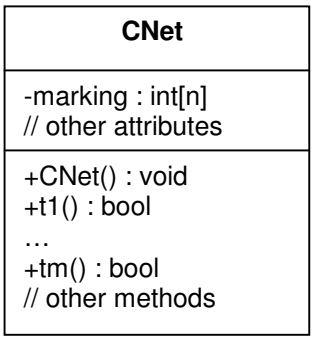


Fig. 19: Net class

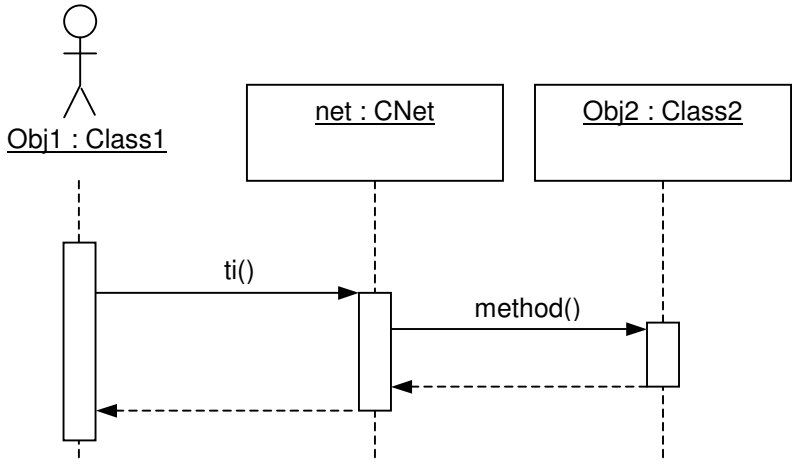


Fig. 20: Transitions triggered by messages

Messages can be received by the net at any time, but they lead to the occurrence of a transition only if it is enabled. In the other case the message will be discarded. The return-value will be true, if the transition is enabled and occurs and false otherwise.

Example: The class for the ccd-net is shown in Figure 21. The attribute vel_cntr holds the control value of the velocity. The methods velocity_control and distance_comp implement the control of the velocity and the comparison of the

measured distance with the computed minimal distance respectively. For better readability the methods have been given names related to the application.

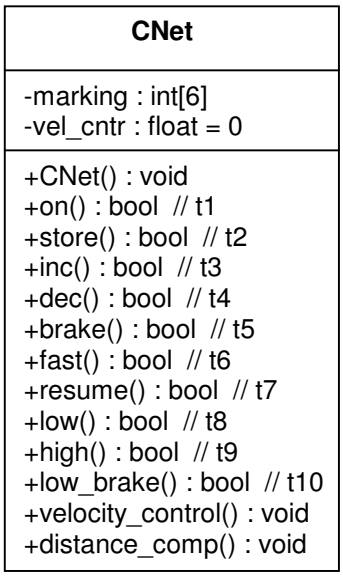


Fig. 21: Class for the ccd-net

We embed the ccd-net into a simulation environment consisting of a simple vehicle dynamics simulation, a buzzer to notify if the stored control velocity is exceeded, a warning lamp to inform the driver that the measured distance is below the minimal distance, a lamp that is switched on if the cruise control is active and a simple user interface.

The sequence diagram in Figure 22 shows the reaction on using the brake pedal. In consequence the velocity is decremented (set_vel_act_brake), the method brake is called, if the cruise control is active, and the state of the activity lamp is set (set_state).

The approaches to code generation from Petri nets can be classified into centralised, decentralised and hybrid approaches [GV03]. In the first approach a centralised scheduler determines which transitions are enabled and dispatches enabled transition sequentially. A disadvantage is that parallelism in the model is not preserved. Furthermore the sequential scheduler forms a bottleneck especially for large nets. On the other hand, the decentralised approach assigns a process to each place and each transition. Parallelism is now preserved at the expense of performance since synchronisation is very time consuming. Combi-

ning both approaches into the hybrid approach means structuring the net into components that should be executed concurrently. The best choice for such components are sequential state machines because they can be easily implemented by a sequential process.

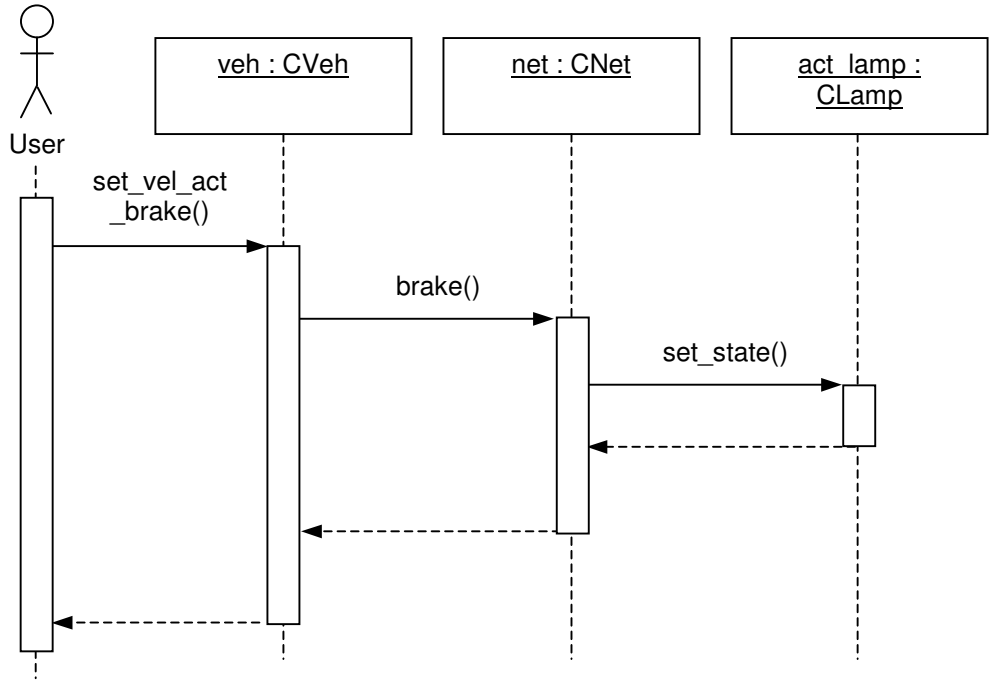


Fig. 22: Sequence diagram for the ccd-net

In our approach we use information from the construction of the net from patterns for the decomposition into components. The branch and merge patterns identify three components, two parallel processes and the process preceding, following the branch respectively. The synchronisation pattern induces that the access to places from different parallel components have to be synchronised, in our approach by semaphores.

Definition: A *component* C of a net $N = (S, T, F, M_0)$ is a subnet $(S_C, T_C, F_C, M_0{}^C)$ where $S_C \subseteq S$, $T_C \subseteq T$, $F_C = \{ (x, y) \in F \mid x, y \in S_C \cup T_C \}$, $M_0{}^C(s) = M_0(s)$ for all $s \in S_C$. A *decomposition* of a net is a set of components such that each place and each transition of the net is element of exactly one component. A place $s \in S$ has to be *protected*, if it is in the pre- or postset of two transitions $t_1 \in T_{C1}$, $t_2 \in T_{C2}$ from different components C_1, C_2. We introduce a semaphore for each place s to be protected which *synchronises* the accesses to the place s of all transitions t for which s is in the pre- or postset of t.

Example: In the ccd-net the transition t_2 (store velocity) implements the branch pattern and thus induces three components C_1, C_2, C_3 where C_2 and C_3 are parallel (Figure 23). These components will be implemented as three processes.

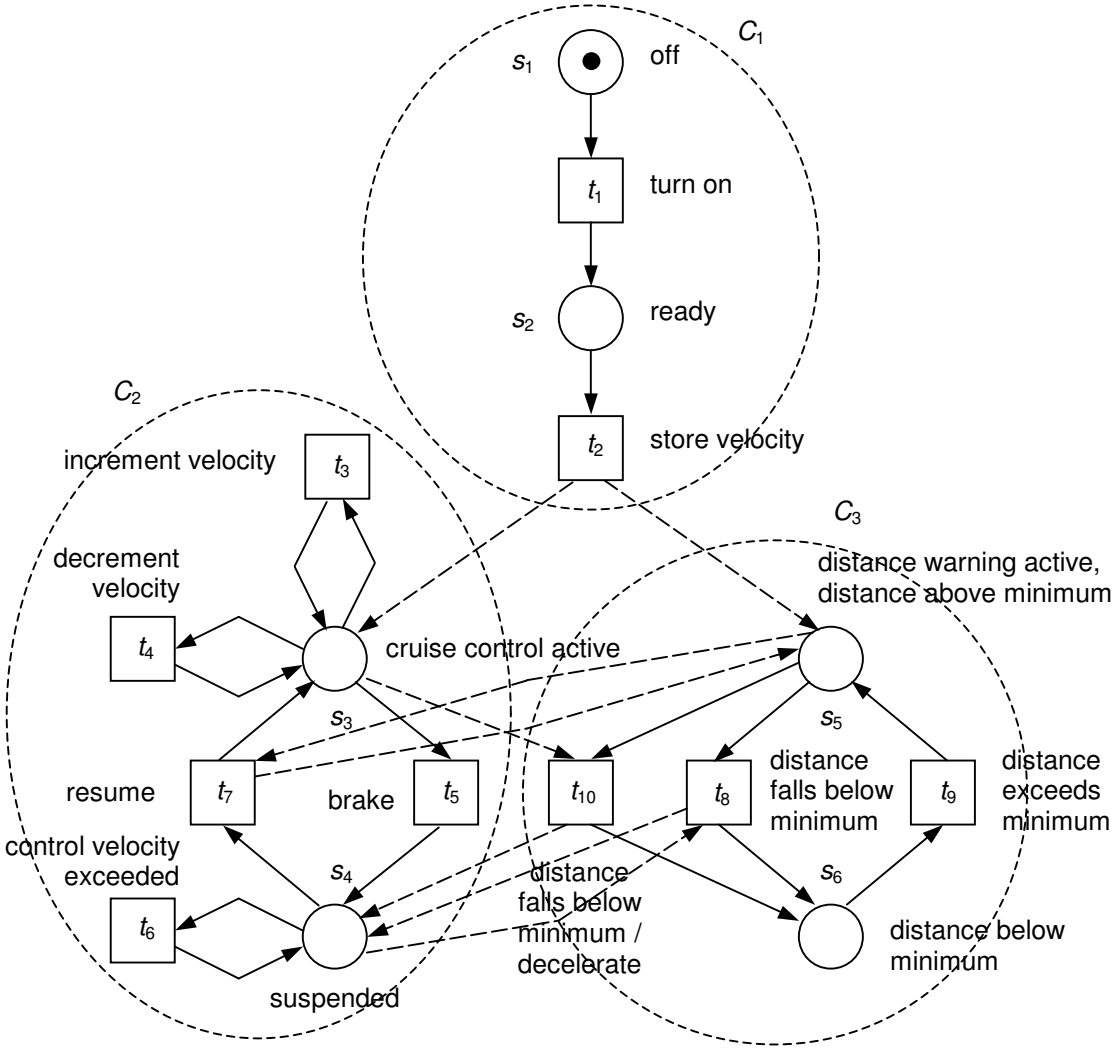


Fig. 23: Decomposition of the ccd-net

The arcs across the border of components (drawn in dashed lines in Figure 23) form synchronisations between these processes. The places s_3, s_4, s_5 involved therein have to be protected by semaphores. Place s_3 is accessed by the transitions t_2, t_3, t_4, t_5, t_7, t_{10}. Therefore we need a semaphore for the synchronisation of these transitions. Place s_4 is accessed by the transitions t_5, t_6, t_7, t_8, t_{10}, which have to be synchronised. Place s_5 is accessed by the transitions t_2, t_7, t_8, t_9, t_{10}. All in all there does not remain much parallelism in the net only between

t_6 and t_9. Note that semaphores must be reserved in a well-defined order, otherwise a deadlock could occur.

The velocity control and the distance comparison work independently and thus can be implemented in two other parallel processes.

A transition is implemented as a method of the class CNet in the following way:

```
bool CNet::t()
{
  bool res;                     // return value: true if transition is
                                // enabled and occurs, false otherwise
  // reserve semaphores for protected places in well-defined order
  Lock(sema1);
  Lock(sema2);
  ...

  // all locks successful?
  if (IsLocked(sema1) && IsLocked(sema2) && ...)
  {
    // transition enabled?
    if (/* preset of transition marked? */)
    {
      // change marking
      ...

      res = true;
    }
    else
      res = false;

    // free semaphores
    ...
    Unlock(sema2);
    Unlock(sema1);
  }
  else
    res = false;

  return res;
}
```

The Petri net implementation is one part of the overall software system. Therefore the test of the integration of the code generated from the net model is one part of the whole software test. We reuse the simulation runs and check the implementation against the model. This means that for each simulation run one or more corresponding test cases are added to the software test specification.

Example: For the ccd-net five simulation runs were specified. For each run a test case is added that triggers the transitions in the order specified in the runs.

6 Conclusion

In this paper we presented an integrated software engineering approach for the usage of Petri nets in software development from analysis to testing of software. The Petri net model forms one part of the software besides other components like user interfaces, data bases, communication routines. Net modules are ideal for the reactive parts of the system such as the control in our example cruise control with distance warning.

References

[AS87] Alpern, B.; Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3): 117-126 (1987)

[Bal98] Balzert, H.: Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, Bd. 2 (1998)

[Bal00] Balzert, H.: Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, Bd. 1, 2.Auflage (2000)

[Bau96] Baumgarten, B.: Petri-Netze. Spektrum Akademischer Verlag, 2.Auflage (1996)

[BG94] Brauer, W.; Gold, R.: Concurrent processes and Petri nets. In: Schwichtenberg, H. (Ed.), Proof and Computation, Springer (1994), pages 1-64

[Boe81] Boehm, B.: Software engineering economics. Prentice-Hall (1981)

[Car02] Carnegie Mellon University, Software Engineering Institute: The Capability Maturity Model Integration, v1.1(2002)

[Ger97] German Ministry of the Interior (Ed.): V-Model. June 1997

[Gol95] Gold, R.: A compositional dataflow semantics for Petri nets. Acta Informatica 32:627-654 (1995)

[Gol02] Gold, R.: Specification and simulation of distributed systems: Process models and dataflow networks. 11th International Colloquium on Numerical Analysis and Computer Science with Applications, Plovdiv, Bulgaria, 2002

[GV03] Girault, C.; Valk, R.: Petri nets for systems engineering. Springer (2003)

[Jal97] Jalote, P.: An integrated approach to software engineering. Springer, 2.Ed. (1997)

[Pnu81] Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science, 13:45-60, 1981

[RR98] Reisig, W.; Rozenberg, G. (Hrsg.): Lectures on Petri nets. Springer, Vol. I: Basic Models (1998), Vol. II: Applications (1998) (Lecture Notes in Computer Science, Volume 1491, 1492)

[Wik97] Wikarski, D.: Petri net tools - a comparative study. Research Reports of FB Informatik, TU Berlin, Report Nr. 97-4 September 1997

Author biography

Prof. Dr. Robert Gold has been holding the lectureship for Engineering Mathematics and Data Processing at Fachhochschule Ingolstadt since 1998.

Gold was born in 1962 in Schrobenhausen/Bavaria. He studied computer science at Technical University Munich, where he also obtained his PhD degree. Before he changed to FH Ingolstadt he worked as a software developer in the automotive industry for Conti Temic and Siemens. Gold has been the dean of the department of electrical engineering and computer science of Fachhochschule Ingolstadt since 2003.

Contact: robert.gold@fh-ingolstadt.de

Impressum

Herausgeber
Der Präsident der
Fachhochschule Ingolstadt

Esplanade 10
85049 Ingolstadt
Telefon: 08 41 / 93 48 - 0
Fax: 08 41 / 93 48 - 200
E-Mail: info@fh-ingolstadt.de

Druck
Hausdruck

Die Beiträge aus der FH-Reihe
"Arbeitsberichte/ Working Papers"
erscheinen in unregelmäßigen Abständen.

Internet
Dieses Thema können Sie, ebenso wie die früheren Veröffentlichungen aus der FH-Reihe "Arbeitsberichte - Working Papers", unter der Adresse *www.fh-ingolstadt.de* nachlesen.

ISSN 1612-6483