

A New Function Block Modeling Language Based on Petri Nets for Automatic Code Generation

Nils Hagge and Bernardo Wagner, *Member, IEEE*

Abstract—A component based modeling language with the component interfaces derived from the elements of the Petri net theory called CNet is presented in this paper. In contrast to IEC 61499 function blocks that define the flow of events and data separately, CNet models them combined as colored tokens. The event-discrete behavior of CNet components is described by a special class of colored Petri nets with timed arcs and sharpened semantics in order to allow automatic generation of possibly concurrent Java code. In this paper, the state space of CNet components is analyzed and a novel event-discrete “handler-based” execution model that implicitly models an automaton is introduced.

Index Terms—Controller design, Java, Petri nets, reusable components.

I. INTRODUCTION

NOWADAYS, distributed control systems are still assembled for the most part from independently designed components without taking into account the interferences of the components’ communication. The degree of complexity of such systems has increased that way that these dependencies between those system parts can no longer be neglected.

CNet [1] offers a solution by designing the whole control system including modeling and analyzing the closed system using the same formalism. The language elements allow modularization and hierarchization and encourage reusing of components. These components are also suitable for describing the plant including the timing of communication paths. In [1], the behavior of all components is expressed exclusively by PNet, which is a special class of colored Petri nets with arc timing, test and inhibitor arcs. The application of languages provided by IEC 61131–3 [7] would also be imaginable, but the strength of PNet is that concurrent runs of events can easily be expressed graphically. In the domain of industrial automation graphical design methods have high acceptance. In order to communicate with the real process plant interfaces are necessary that are able to serve as a replacement for the model of the process. In IEC 61499 function block language [8], special service interface function blocks (SIFB) with some predefined inputs and outputs serve as link to the process. In CNet the interfaces of all components follow the same formalism without any restrictions no matter if a component represents a control algorithm, a means of communication, or a model of the plant. In [9], a mapping from the formal elements to a representation in the Java programming language is proposed. For IEC 61499 functions blocks (FB) and

other formalisms such a mapping does not exist yet. The creation of such had the outcome that parts of a control system designed using these formalisms could interface to CNet models on Java programming language level. Different mappings can be managed by what is known in software engineering as the “adapter pattern” [6]. This also helps building bridges for heterogeneous systems interfacing between different technologies [10].

In Section II, an overview of CNet will be presented before focusing on the interface elements of CNet with its semantics in brief in Section III. Section IV compares the event and data connections of the IEC 61499 function blocks with their representation as typed tokens in CNet. Section V explains how PNet defines the event-discrete behavior of CNet components. It is further discussed how determinism with regard to modeling real-time applications is achieved. The section recalls the example controller from [9] and performs a state space analysis. Its results will lead to a new way of implicitly representing a CNet components’ state space. This representation is shown in Section VI. It allows a direct mapping to a very efficient concurrent implementation that shall be called “handler-based execution model.” Section VII presents an example of a process component and its corresponding driver in Java that realizes the actual hardware access. The paper finishes with a conclusion.

II. CNET OVERVIEW

CNet is short for “component net” and is based on the work of Wurmus, who first proposed in his dissertation thesis [1] the idea of using the elements of the classical Petri net theory as interface elements of hierarchical composable components. CNet differs from other hierarchical approaches (e.g., [2], [4]) in the way that it uses places and transitions as interface elements, but only places for inputs and transitions for outputs.

The basic idea behind CNet stems from the observation that classical huge Petri net designs lose the original clarity and conciseness of the powerful and limited number of graphical elements. On one hand, CNet lets huge designs reconquer the clarity of a graphical representation by clustering special functional parts with strongly connected elements and identifying them as components by giving them self-explanatory names. This hides complex implementations and simplifies the understanding of the whole model. On the other hand, it offers the possibility to reuse parts of already implemented functionality by instantiating one component two or more times, which also simplifies a net model and eases the design process.

A. Components and Ports

Several approaches to hierarchize Petri nets have already been published [2]–[5]. In [4], hierarchy is expressed, e.g.,

Manuscript received March 7, 2005; revised June 14, 2005.

The authors are with the Institute for Systems Engineering, Real-Time Systems Group, University of Hanover, D-30167 Hannover, Germany (e-mail: hagge@rts.uni-hannover.de; wagner@rts.uni-hannover.de).

Digital Object Identifier 10.1109/TII.2005.857614

by so-called substitution transitions on higher-level nets called pages. These special transitions represent subnets, or in bottom-up view, hierarchization is done by hiding a part of a net and representing this part by a single net element. This idea can be multiply applied on a net and produces several levels. The disadvantage of this technique is that the original Petri net semantics are not preserved except for the deepest level: the classical firing operation of a transition atomically consumes tokens from its pre-places and puts new tokens in its post-places. This means that the total number of tokens stays constant for the case that the sum of all pre-arc weights is equal to the sum of post-arc weights. In the case of a hierarchically refined transition, this property no longer holds, since tokens may disappear and reappear. Furthermore, the components that we have in mind to represent controllers and parts thereof need interfaces that are comprehensible without seeing their internal implementation. In software engineering it is generally accepted to separate interface definitions from concrete implementations. In contrast to this, Jensen *et al.* in [4] do not provide separate formal interface definitions. Their hierarchies first must be flattened in order to get a formally analyzable colored Petri net. Additionally, the process of substitution does not only replace the corresponding substitution transition, but also modifies peripheral net elements. This means that in top-down refinement and in bottom-up composition no implementation hiding is possible and that going one level up in hierarchy not only simplifies the view to the model, but also reduces its formal preciseness. Another concept presented in [4] is the means of defining fusion sets. The approach in [3] refines this idea in order to allow fully-fledged Petri nets to define components by using solely transitions as synchronization points. Connecting components is realized by fusing the transitions that form the synchronization points, i.e., each connection is represented by a transition that belongs to two or more components. On a distributed system, it is difficult to implement a firing operation while keeping its atomic character, when at least two of the components are situated on different devices. Jensen [4] discusses the concepts analogously for places. CNet provides means for exact interface descriptions that can be independently defined from any implementation. These interfaces consist exclusively of *places* as *inputs* and *transitions* as *outputs*. As explained in [9], this arrangement has two profound advantages over the other possible combinations. First, components can simply be connected by post-arcs, no additional basic elements are needed as “glue” as would be the case if, e.g., inputs *and* outputs were represented by transitions, because at least one additional place would be necessary, since transitions may not be connected directly. Second, the neutrality of unconnected inputs and outputs, which is a key feature for reusability of components. Normally, one wants to design reusable components as general as possible. This means that the user of such a component might not need the whole functionality of a component and leaves some ports unconnected. Neutrality means that unconnected ports do not affect the rest of the component. This is guaranteed when inputs are modeled as places and outputs as transitions, since unconnected inputs appear as places that are never marked and unconnected outputs as transitions that immediately fire, when all internal preconditions are met. In other words, the part of the net connected to the unused ports will not block the rest of the component.

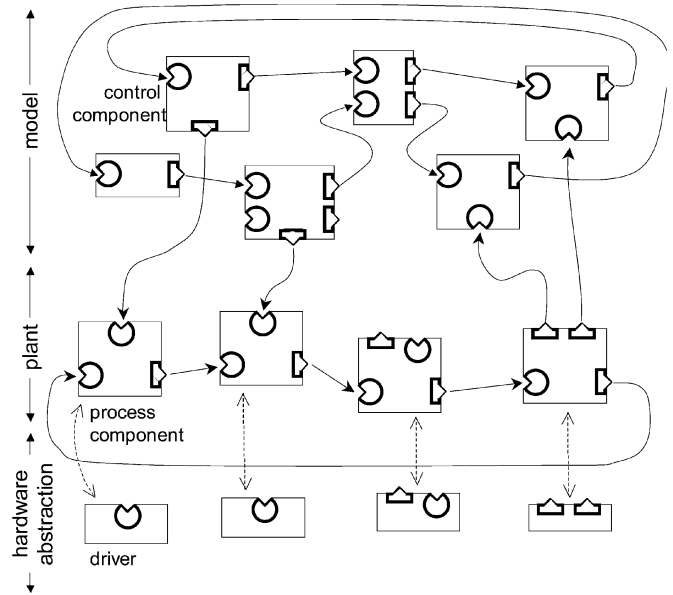


Fig. 1. CNet design with control and process components and drivers as abstractions of the hardware interfaces.

B. Component Types

All component interfaces consist of the same elements, although the components can be distinguished into control components, process components, and drivers (see Fig. 1). The *process* components model the behavior of the uncontrolled plant, while the *control* components implement the control algorithm. For the installation of the design onto the controller hardware only the control components are relevant. The process components are for offline simulations and/or analysis, since they are a model of the plant operations. Connections between process and control component shall be called *vertical* connections, while connections between components of the same type are named *horizontal* connections. The vertical connections depict the interaction of the controller and the process.

C. Mapping

Before the design can be installed onto the controller device or the controller devices, if the system is distributed, the different components have to be mapped to a specific device. The process components are replaced by driver stub components. These show the same vertical interface as their corresponding process component and encapsulate the platform-specific implementation for the hardware access to the real process element, i.e., the sensor or actuator. The process components or driver stubs, respectively, can only be placed onto devices where the appropriate hardware access can be realized. The control components on the other hand can in general be put onto any device of a distributed system connected by a network. An advantageous mapping places control components onto the same device where the driver stubs it communicates with are located. One possible assignment of components to devices is shown in Fig. 2 (see dashed encircling).

D. Communication

In a distributed system, it is unavoidable that components that are situated on different devices have to communicate with each

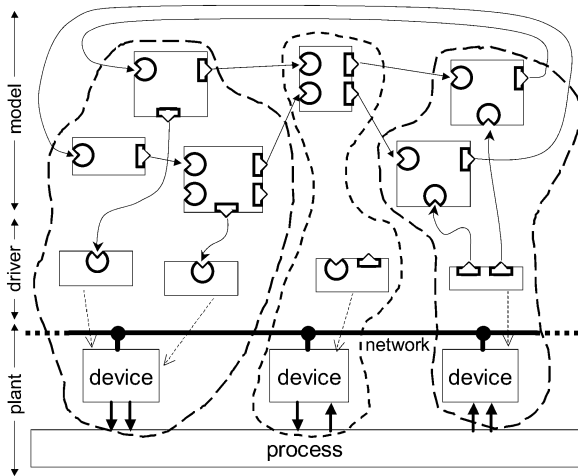


Fig. 2. Replacement of process components by driver stubs and mapping to devices of the plant.



Fig. 4. CNet interface elements a) input place b) output transition.

test and inhibitor arcs. Its language elements cover means to express events, structured data, delay times, concurrency, and synchronization. For the formation of the interfaces of CNet components the two new Petri net elements, *input place* and *output transition*, (see Fig. 4) have been created [1]. CNet/PNet only covers a small number of (new) language elements in order to be quickly and intuitively comprehensible. Furthermore the extensions introduced by CNet/PNet do not destroy the formal analyzability, which is a powerful means to decide on the correctness with respect to determinism, deadlocks, and boundedness of the modeled controller. Although some of these algorithms are not directly applicable to CNet components, as they do not represent closed-loop nets, Section V will show how the standard algorithm for the reachability analysis is extended to be applicable to CNet.

A. CNet Interface Elements

The two main interface elements of CNet components are as mentioned before the input place and the output transition (see Fig. 4). These elements are derived from the corresponding elements from the Petri net theory. This ancestry also leads to a clearly and comprehensibly defined behavior of inputs and outputs of CNet components concerning the event handling, which will be explained in the following.

B. Events

The arrival of an event at a component's input is modeled by placing a token into it. This has further implications on the way CNet components recognize and process events. Since a token in an input place does not disappear unless consumed by a firing internal transition and assuming a capacity of one for the input place and following the strong firing rule a second token may not arrive unless the first is processed which has to be guaranteed by the implementation, i.e., in CNet events must always be handled some way and thus are not unintentionally ignored. A capacity greater than one can be assigned to an input place, so that it will act as an event-queue. Alternatively, an arrangement of one internal place and one transition in conjunction with the appropriate arcs can be used to store the latest event only (cf. design pattern "actual value" [1]). The output transition generates an event in terms of a token, when all internal pre-conditions (and possibly post-conditions) are met. The token is only generated, if the output transition is connected to something, i.e., an input place or a regular place.

C. Data

The previously described events can be parameterized with ancillary data, which on one hand makes events distinguishable and on the other hand allows carrying special process information. This is realized by so-called "colored tokens" [4]. The term "color" refers to the distinguishability of the tokens, but simply stands for a specific value out of a defined value set, called "color set." The concept is the same as in typed structured programming languages (e.g., Pascal). PNet offers enumerations, integer

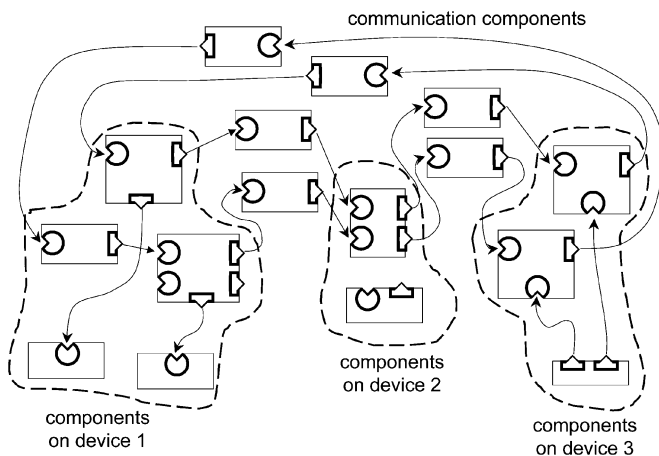


Fig. 3. CNet design after mapping of components to the devices and insertion of communications components.

other over a network. This means that the connections realized by arcs going from one encircled group of components to another (Fig. 2) cannot directly be realized. For this reason these connections will be disrupted and an additional component each will be inserted (see Fig. 3). This communication component encapsulates the access to the network and can be considered a pipe whose ends are each a part of the appropriate component cluster. The connection parameters are implemented as component parameters. Like process components communications components also feature at least two realizations: the driver that implements the real access to the network and a behavioral implementation that takes the transmission latencies into account. Different driver implementations for the same behavioral component allow using different kinds of physical network [10], e.g., the behavior of a periodic data transmission can be realized by a time-triggered field bus or RTnet [12].

III. CNET/PNET

The internal behavior of CNet components can be completely expressed by PNet and nested CNet component instantiations. PNet is a special class of colored Petri nets [4] with timed arcs,

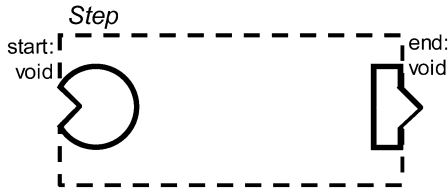


Fig. 5. CNet interface for an execution step.

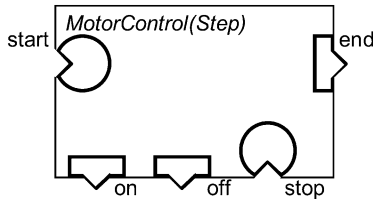


Fig. 6. Component implementing the step interface.

subranges and records as types (cf. Pascal). Records may also be nested. The special type *void* denotes that no information is available. The type *any* specifies that any type, i.e., color set, is accepted in a specific situation and that the actual information is left unconsidered at that place.

D. Time

Time is not an interface element, but is strongly related to the event generation. Each token in CNet/PNet including those of color set *void* possesses a timestamp, i.e., the absolute time of the execution platform at the moment, when the token is created. There are two possibilities to create tokens: at system start-up, when the initial marking is set-up, or under operating conditions each time a transition is fired. This includes the moment, when a token is put into an input place as result of a firing transition. The timestamp of a token can be included in the firing conditions by means of timed arcs. Timed arcs define lower and/or upper bounds to the duration that an individual tokens may reside on a place in order to enable a transition. This is a convenient feature for modeling timers, timeout watchdogs etc. In IEC 61499 special function blocks are needed for this purpose.

E. Interfaces

A number of interface elements can be grouped together as a CNet interface (Fig. 5) in order to represent certain functionality or service [9] that is provided by a component implementing it (Fig. 6). As already mentioned the implementation of a component may use other nested components internally, which is not visible from outside. It is also possible to design components that manage certain kinds of sub-components. The type of sub-components is expressed by the required CNet interface. This information is made public and becomes a part of the component interface. The scheduler component in Fig. 7 provides two “sockets” for components implementing the step interface to be plugged in, e.g., two instances of *MotorControl* (Fig. 6).

IV. COMPARISON WITH IEC 61499

In this section, an example of a composite function block according to IEC 61499 is considered and compared to a reimplementation with CNet language elements. This shall on one hand

side help to understand more clearly how CNet can be used, assuming that the reader has basic knowledge of IEC function blocks and, on the other hand side, shall demonstrate that modeling events and data as one entity by typed tokens as PNet does avoid possible design errors causing data inconsistencies based on unconsidered dependencies between data and event connections.

A. Short Introduction to Function Blocks

The graphical symbol used for the function blocks with its interface elements defined by the IEC 61499 [8] is depicted in Fig. 8. The upper part serves as interface for the events: the left side takes the event inputs; the event outputs are on the right. The lower part is responsible for the data ports. The internal realization of the behavior is hidden. Paragraph 1.4.5.3 of [8] describes temporal dependencies that have to be taken into account in order to design a correct and compliant execution environment for function blocks. The most important idea is that when event inputs are triggered the corresponding data must already be stable at the corresponding data inputs. The dependency between an event and its data is expressed by the *with* clause in structured text or graphically by a vertical line with squares marking the appropriate data and event ports (see Fig. 9). Fig. 9(a) shows the example of a function block named *PI_REAL* with the event inputs *INIT* and *EX*, the event outputs *INITO* and *EXO*, the data inputs *HOLD*, *PV*, *SP*, *KP*, *KI*, and *CYCLE*, and the data output *XOUT*. The type of each port is stated on the outer left or right side, respectively. The *EX* input event e.g., processes all input data. *INIT* is not associated with data, while *XOUT* is valid on both output events as indicated by the vertical line and the squares as described above.

B. Translating the Example Into CNet

The same interface representation can be achieved with CNet as shown in Fig. 10 with the restriction that CNet normally does not provide the primitive data types *REAL* and *TIME*.

The left part of Fig. 10 shows the graphical representation of the CNet interface definition. It includes two input places and two output transitions. Input places correspond to the event inputs of IEC 61499, and therefore in this example they also wear the names *INIT* and *EX* from Fig. 9. The output transitions are named *INITO* and *EXO* correspondingly. The data ports do not appear explicitly in CNet. Data is modeled as information tagged to a token. The arrival of a token at an input place corresponds to the triggering of an event input of a function block. This way of modeling events and data together is simpler than having to define two ports and connections and completely avoids the consistency problems that may arise from not respecting required temporal and logical dependencies between events and data. The concept of typed tokens binds data undividable to the corresponding event. The function block *PI_REAL* [Fig. 9(a)] requires valid data at all inputs on appearance of event *EX*. In other words event *EX* is parameterized by *HOLD*, *PV*, *SP*, *KP*, *KI*, and *CYCLE*. In CNet this is expressed by a typed token. Input place *EX* is typed with *C1*. *C1* is a record data type that results from an aggregation of primitive data types as shown on the right side of Fig. 10. The *INIT* event has no associated data and is therefore indicated by untyped tokens. The output

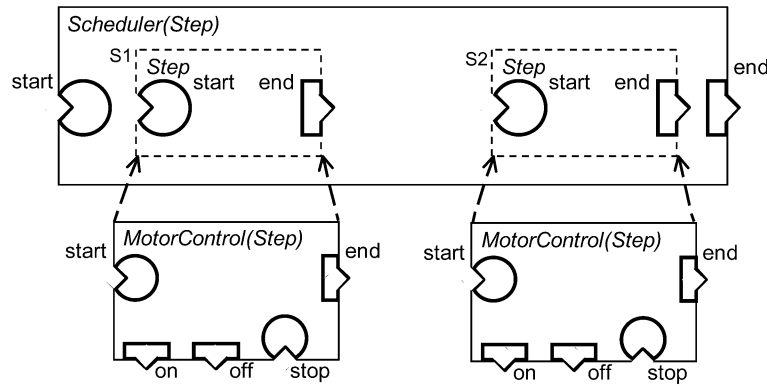


Fig. 7. Component that requires two depending components implementing the step interface.

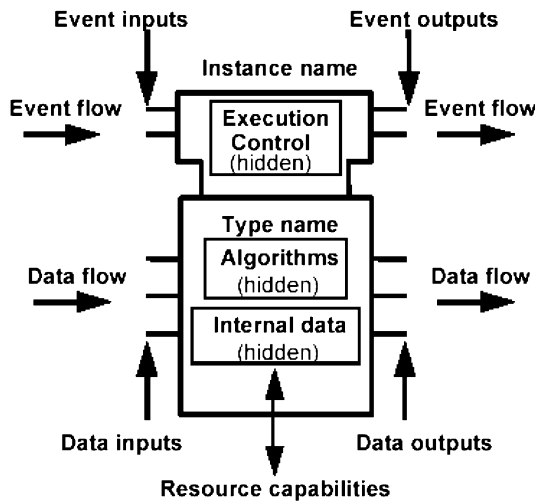


Fig. 8. Function block with interfaces [8].

events INITO and EXO each produce values of type REAL. Correspondingly, the output transitions are of type REAL indicating that they generate tokens tagged with real values.

Fig. 11 presents a realization of function block PI_REAL as composite function block. It has been composed of one instance of function block PID_CALC and one instance of INTEGRAL_REAL. In the following this realization shall be reimplemented using CNet.

The first step for the definition of the appropriate CNet interfaces is to find the correct event-data associations of the subblocks. These are not shown in Fig. 11. They can only be taken from the definition of the function block in Fig. 12. This leads directly to the CNet interface in Fig. 13. For function block PID_CALC a corresponding CNet interface is found the same way. When trying to translate the internal arrangement from Fig. 11 into CNet/PNet, one realizes that the connection from event output CALC.INITO to event input INTEGRAL_TERM.INIT has no counterpart in PNet, since the output transition created for CALC.INITO and the input place for INTEGRAL_TERM.INIT (see Fig. 13) are neither syntactically nor semantically compatible, because only the event consumed by input INTEGRAL_TERM.INIT originates from function block CALC, the associated data value CYCLE comes from the outer interface. This problem could be solved by the interposition of an additional place for joining the event with the previously stored information.

Deeper consideration reveals that even this is not possible in this case, since the outer interface (of PI_REAL) does not sample any value for CYCLE on event INIT that could be stored. In fact, the original arrangement in Fig. 11 taken from [8, p. 31] contains the design fault that function block INTEGRAL_TERM reads in a data value on event INIT that is undefined during initialization and not valid before first arrival of event EX [see event-data associations in Fig. 9(a)].

C. Correction of Original Example

In order to proceed with the translation of the original function block example, the outer interface of PI_REAL is modified as shown in Fig. 9(b) such that on event INIT the needed data value for CYCLE is sampled. This enables the full PNet implementation of PI_REAL to be completed as depicted in Fig. 14. With the modified interface the component requires the tokens placed at input INIT to be tagged with a value of type TIME. The value is bound to the internal variable t and intermediately stored in an auxiliary place of type TIME as described before, since the subcomponent takes only untyped tokens at its INIT input. When PID_CALC completes its initialization, it produces a real value tagged token at INITO. This token drops its value and is joined with the previously stored TIME value. This new token correctly triggers the initialization of INTEGRAL_TERM. The remaining part of the PNet implementation works in a similar way.

Most of the PNet elements and inscriptions surrounding the embedded components in Fig. 14 are needed only for “type casting” purposes due to different colorsets used at the outer and inner interfaces. A redesign of PI_REAL with the same functionality but improved sub-component interfaces with adjusted types is given in Fig. 15.

D. Conclusion of Comparing CNet to IEC 61499

The example has shown that modeling events and data together as one entity avoids design faults causing data inconsistencies. The problem arose from the arrangement in Fig. 11 where the data-event dependencies are not visible, but existent. The way CNet combines events and data forces these dependencies to be respected.

The following section will explain the possibilities that PNet offers and will define the behavior of CNet components in more detail.

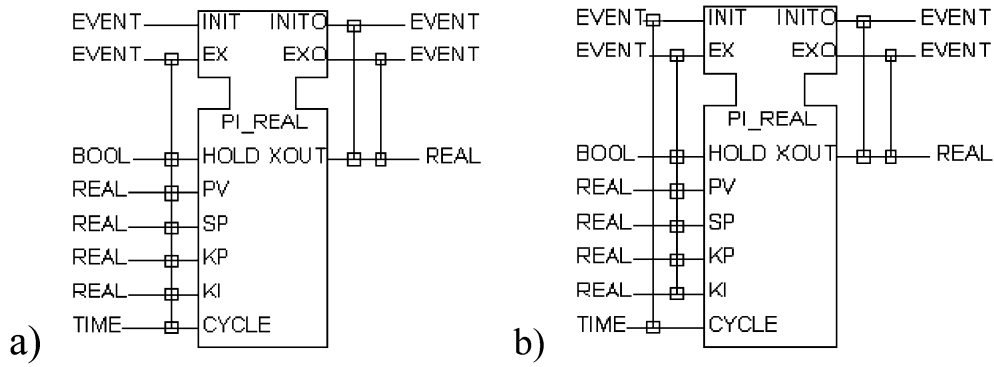


Fig. 9. Example function block PI_REAL [8] a) original b) modified.

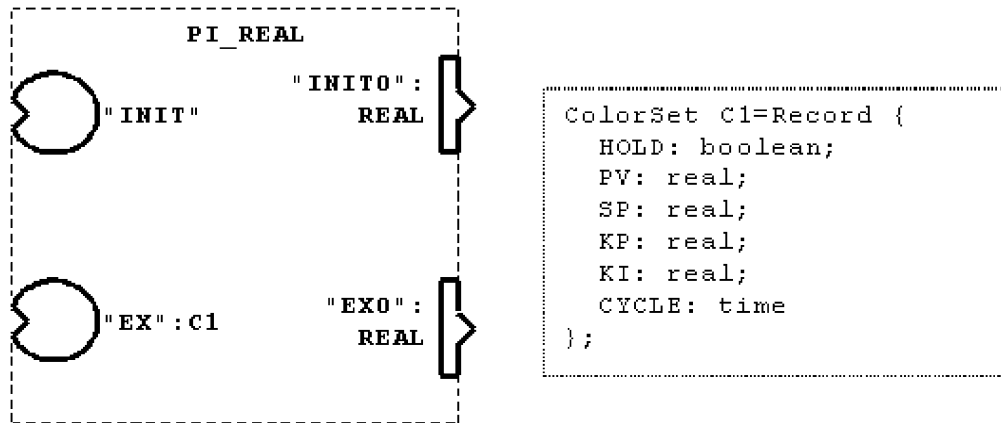


Fig. 10. Definition of function block PI_REAL as CNet component.

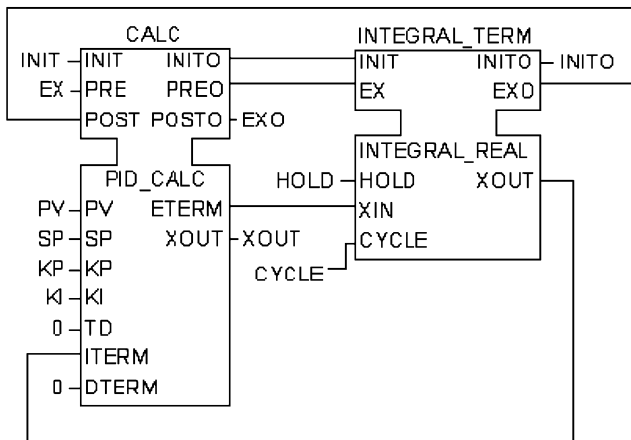


Fig. 11. Internal arrangement of PI_REAL as composite function block.

V. BEHAVIORAL DESCRIPTION WITH PNET

As outlined in previous sections, the behavior of CNet components is expressed using PNet. An implementation of component MotorControl (Fig. 6, [9]) is shown in Fig. 16. This simple component is suitable for starting a motor on request and stopping it on reception of another signal (cf. e.g., the motor that opens a lift door). Details about repetitive start requests while running are explained in [9]. This model uses only undistinguishable tokens, i.e., tokens of color set *void* in terms of PNet. The PNet model consists of four standard places, namely *ready*, *running*, *on*, *off*, and *s7*, of which the first one is initially marked,

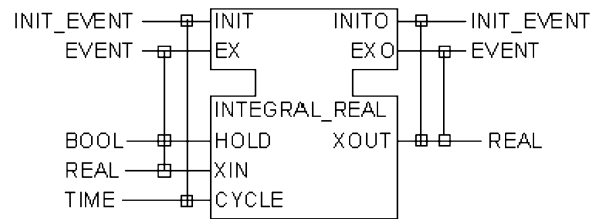


Fig. 12. Interface definition of function block INTEGRAL_REAL.

the input places *start* and *stop*, the standard transitions t_1 and t_2 , and the output transitions t_3 , t_4 , and t_5 , and arcs as shown. PNet transitions are required to fire as soon as possible. This means that a transition whose firing conditions are met must fire immediately. In the case of t_1 in Fig. 16 this means that the triggering of input *start* results in the firing operation of t_1 . This firing operation is atomic and consists of removing the token from *start* and *ready* and placing a new one into *running* and *on* each. If, in the situation shown in Fig. 16, input place *stop* were triggered, only a token would appear in *stop*. Transition t_2 could not fire since *running* is not marked and *ready* is not free assuming a default capacity of one for all places. In the situation where an input place is marked and its capacity is exhausted, further triggering of that input is an illegal operation. This can be compared to setting both inputs of an RS flip-flop to logical high value. The enclosing circuit has to ensure that this input pattern is avoided. In the same way a CNet implementation has to enforce that the capacity of input places is not exceeded, so that tokens do not get lost. This requirement is expressed by the *strong firing rule*.

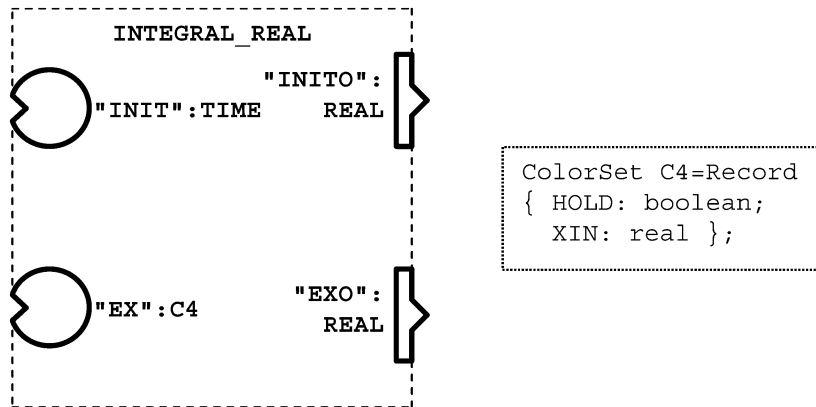


Fig. 13. CNet interface definition for INTEGRAL_REAL.

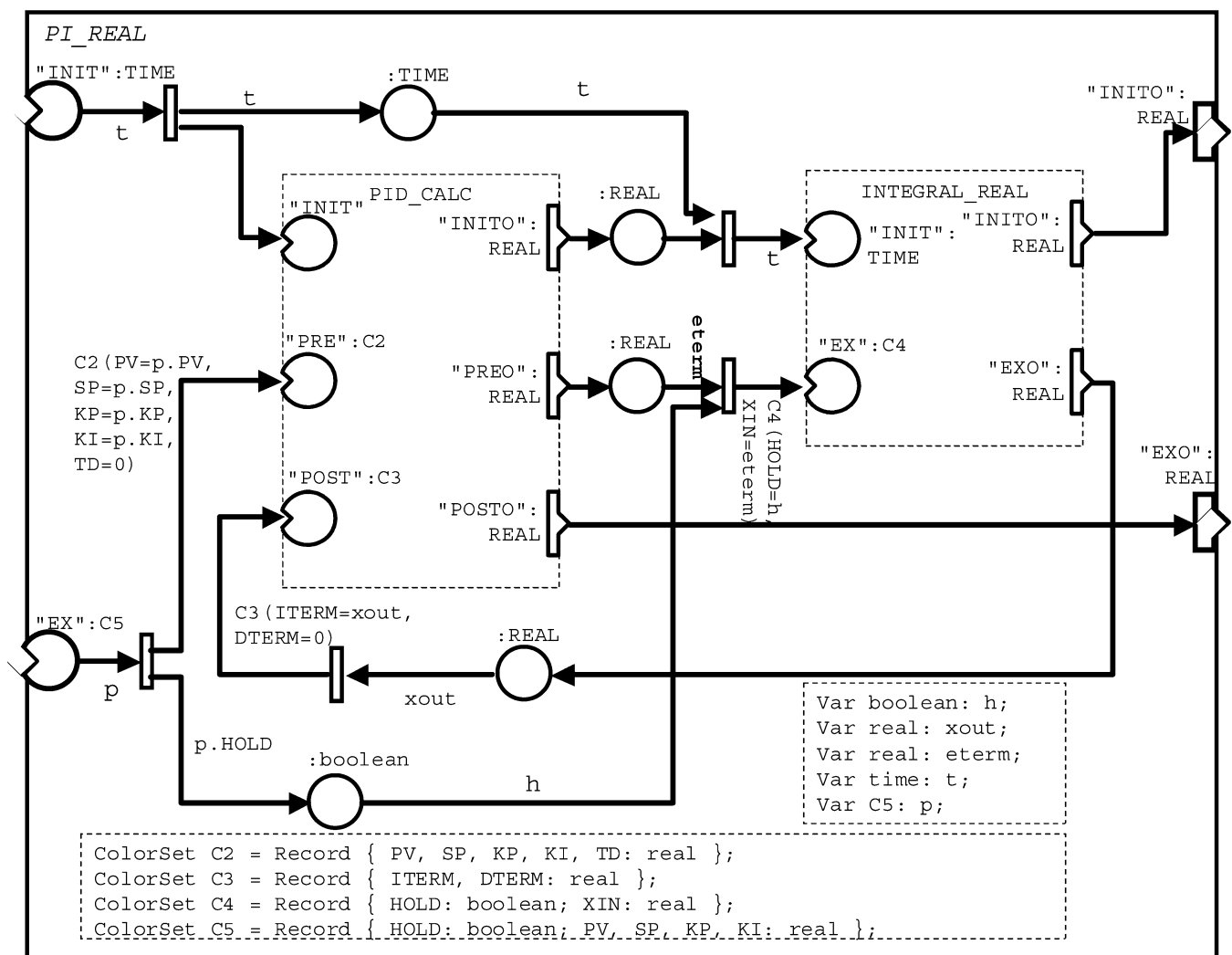


Fig. 14. PNet implementation of PI_REAL with corrected behavior.

A. Extended Reachability Graph

In general, analyzing Petri net designs of real-world applications gets very calculation intensive because of state-space explosion as soon as the net models become bigger. In CNet, every component can be analyzed separately. This divide-and-conquer approach significantly reduces computational effort and

produces robust and reusable components with well-defined behavior that does not have to be reanalyzed when the components are used in new designs. The standard procedure to construct the reachability graph of classical Petri nets [11] has been extended for the analysis of CNet components. The considerable modification consists of treating the marking of input places as special firings in addition to the firing operations of transitions. In order

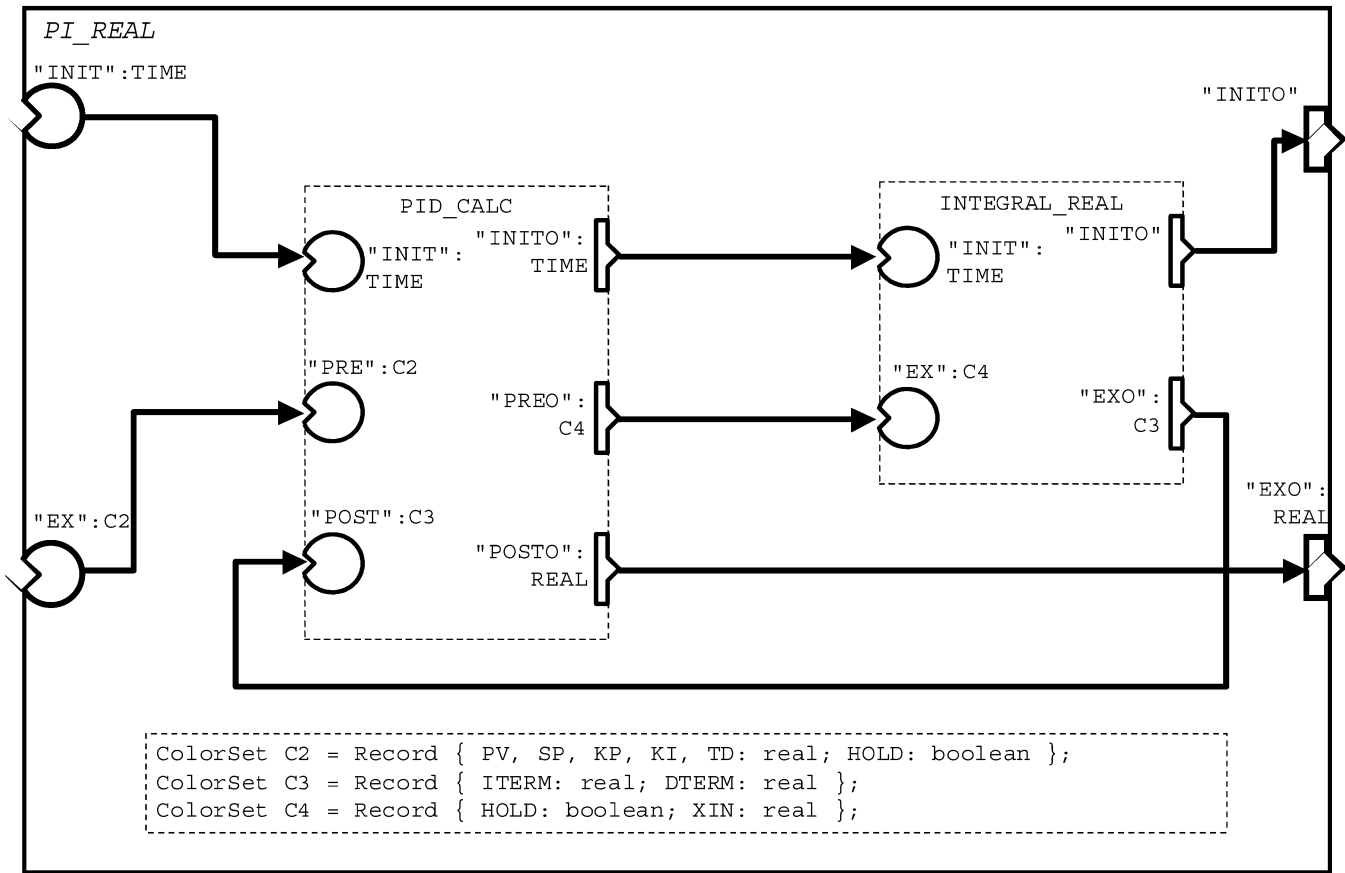


Fig. 15. Redesign of PI_REAL with adjusted interfaces.

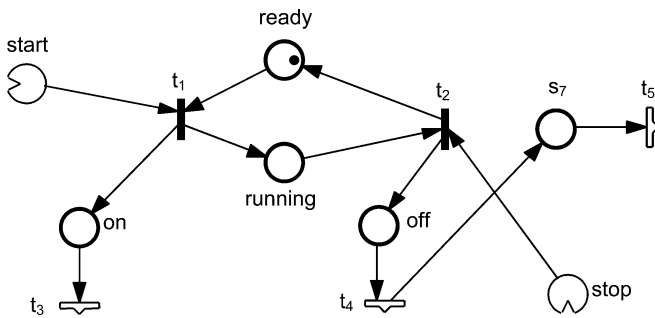


Fig. 16. PNet implementation of component MotorControl.

to further reduce the complexity of the resulting graph, the firings of transitions are given priority over the marking of an input place. This restriction turns out to be another advantage, since it implements the arrangement that external event are only recognized by a component when all internal events are processed. Furthermore, this allows the distinction of *stable* and *transient* markings. In other words, a CNet component is only able to recognize events at its inputs when it is in a stable state and accepts only one event at a time. This is a generally accepted way to handle event-discrete behavior.

The algorithm for the graph construction is shown in Fig. 18. The input data are the set of all transitions T , the set of all input places E , and the initial marking m_0 . The result is the graph that appears as set of its nodes A , representing all reachable markings, and set of edges B , that contains all possible transitions

from one marking to another. The operations $m \circ t$ and $m \circ e$ describe the markings that result from firing transition t and triggering input e , respectively.

B. Interpretation of Reachability Graph

The result for the component MotorControl is presented in Fig. 17. The *stable* markings are depicted as rectangular nodes while the *transient* ones appear as circles. As noted below, there is a third type of marking that appears in the context of timed arcs: *quasistable*. The closed path $0 \xrightarrow{start} 1 \xrightarrow{t1} 3 \xrightarrow{t3} 5 \xrightarrow{stop} 8 \xrightarrow{t2} 11 \xrightarrow{t4} 14 \xrightarrow{t5} 0$ is the deterministic firing sequence of the component as long as the inputs *start* and *stop* are triggered alternately and starting with *start*. This sequence is what the designer might have intended as the exclusive use case for the application of the component. The construction of the extended reachability graph reveals in this case that there are additional nondeterministic paths through the graph that can be taken, if the inputs are triggered in a different order. This is no problem if the enclosing controller ensures that this cannot happen so that these paths are never taken (cf. again forbidden input pattern of RS flip-flop).

An improved version of the component MotorControl that consumes stop requests when it is not in running state and start requests when not in ready state is shown in Fig. 19. The corresponding reachability graph is given in Fig. 20. The graph shows a total of nine possible markings including the two necessary stable markings in contrast to 21 total and four stable markings in the previous version.

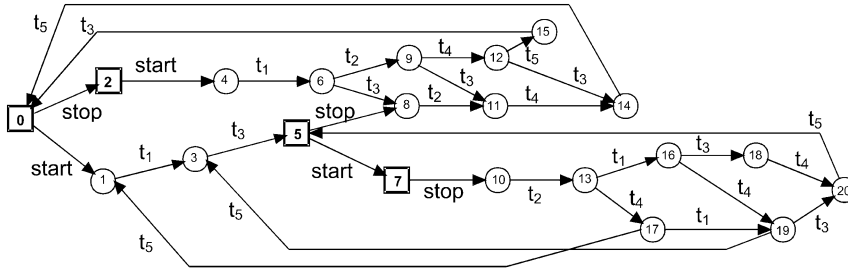


Fig. 17. Extended reachability graph for PNet implementation of CNet component MotorControl.

```

A = {m0}
B = {}
T = set of all transitions
E = set of all input places
Repeat
  foundnode := false
  For all m ∈ A
    fired := false
    For all t ∈ T
      If t is enabled on m
        m+ = m ∘ t
        If m+ ∉ A
          foundnode := true
          A := A ∪ m+
          B := B ∪ (m, t, m+)
          fired := true
    If ¬fired
      For all e ∈ E
        If e is triggerable under m
          m+ = m ∘ e
          If m+ ∉ A
            foundnode := true
            A := A ∪ m+
            B := B ∪ (m, t, m+)
  Until ¬foundnode

```

Fig. 18. Algorithm for construction of the extended reachability graph of a CNet component.

C. Timed Arcs

An example implementation using timed arcs for timeout detection is shown in Fig. 21. Triggering input place p_1 starts the “timer.” If during the following 2000 units of time input p_2 is also triggered, transition t_2 will fire, transition t_1 indicating the timeout otherwise.

The extended reachability graph is depicted on the right of Fig. 21. It contains three stable markings (nodes 0, 1, and 2), one of which is *quasistable* (node 1). Quasistable markings represent states that are left either by external events (here: triggering input place p_2) or automatically by expiration of a timer (here: firing of t_1).

VI. IMPLICIT STATE SPACE REPRESENTATION

The previous section has explained how the behavior of CNet components is defined by means of PNet elements. This section introduces a new way of representing the state space of controller components with concurrent behavior particularly with regard to an efficient, concurrent, and deterministic execution.

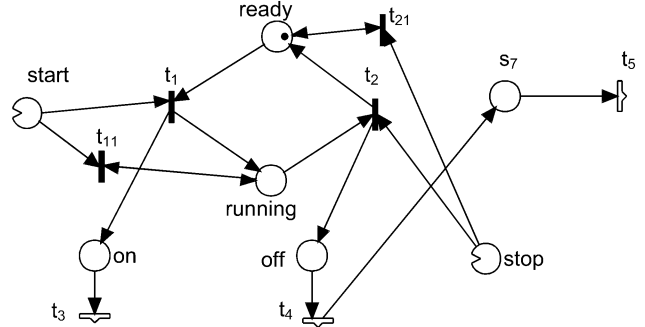


Fig. 19. Alternative implementation of MotorControl ignoring out-of-order triggering of the inputs.

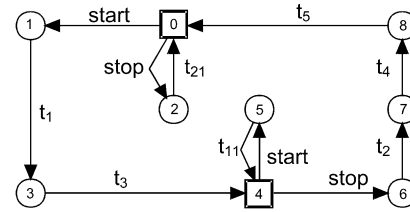


Fig. 20. Extended reachability graph of alternative implementation.

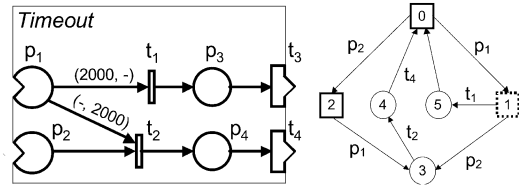


Fig. 21. Component implementation using timed arcs, reachability graph.

It will be shown how the specification of a CNet component can be transformed into executable code. The development of a “virtual machine” for CNet and PNet would be one way to execute component specifications in the context of control mechanism. This direct interpretation of the net models leads to many repeatedly checked firing conditions, where the vast majority of evaluations will fail and have to be rechecked. This results in unproductive processing time, which shall be avoided, especially in power constraint embedded devices. In other words, there are only a few transitions that can fire. Additionally, in the event-discrete domain, a state transition can only take place if and only if an event has happened. First, this means that if no events arrive, absolutely nothing has to be done; the processor of the controller will be idle. Second, the construction of the extended reachability graph as shown in the previous section identifies all possible markings (i.e., states) and events that are

relevant for these states. The total number of events that are processed by a component (or a system) is always finite. This is the basic idea, from which the implicit state space representation that is presented here, is derived from. The finite set of events includes all possible state changes evoked by triggering input places and expiration of internal timers that result from timed arcs. This set E depicts the input alphabet of the controller

$$E = \{e_0, e_1, \dots, e_{n-1}\}. \quad (1)$$

An *action* describes a “short” algorithm that is associated with an event in order to perform the state transition, when the event happens. Set A contains all actions needed by the considered component or system:

$$A = \{a_0, a_1, \dots, a_{n-1}\}. \quad (2)$$

The current assignment of an action $a \in A$ to an event $e \in E$ shall be called a *binding*. The aggregation of all bindings can be expressed as function

$$b: E \rightarrow A \quad (3)$$

that assigns exactly one of the actions to each individual event. It has to be emphasized that these bindings are not static, but change dynamically when events happen by the application of their currently assigned actions. In this way, the current binding function b is an implicit representation of the current state of the considered component or system.

A. Event Definitions

For the timeout component in Fig. 21 set E is disposed to

$$E = \{e_1, e_2, e_t\}. \quad (4)$$

This means that the component timeout can process three different events. This includes triggering one of the two inputs (event e_1 and event e_2 identify the activation of input p_1 and p_2 , respectively) and the expiration of the internal timer (event e_t) that is started after arrival of event e_1 . These events have to be bound to actions that reflect the behavior of the component in its current state. For the initial state following settings shall apply

$$\begin{aligned} b_0(e_1) &= a_{1first} \\ b_0(e_2) &= a_{2first} \\ b_0(e_t) &= a_{error}. \end{aligned} \quad (5)$$

This defines the actions a_{1first} , a_{2first} , and a_{error} . Action a_{1first} realizes the operations that must be executed so that the entirety of the bindings reflects the state of having received event e_1 . This corresponds to the state transition depicted in the extended reachability graph of Fig. 21 as directed arc from node 0 to node 1. Accordingly, the action a_{2first} has to implement the transition from node 0 to node 2 for the appearance of event e_2 in the initial state. The event e_t cannot occur in initial state, therefore the error action is bound to it at this point. Such actions have to be defined for all stable and quasistable states found in the ex-

tended reachability graph. For the analysis of node 1 (Fig. 21) this leads to the following bindings

$$\begin{aligned} b_1(e_1) &= a_{error} \\ b_1(e_2) &= a_{2after1} \\ b_1(e_t) &= a_{timeout}. \end{aligned} \quad (6)$$

Node 2 of the reachability graph models a situation, in which only event e_1 is accepted:

$$\begin{aligned} b_2(e_1) &= a_{1after2} \\ b_2(e_2) &= a_{error} \\ b_2(e_t) &= a_{error}. \end{aligned} \quad (7)$$

The actions itself are considered in the following. Their primary purpose is to implement the dynamic modifications of the bindings resulting from the state change triggered by the specific event.

B. Action Definitions

The initial state is described by the bindings given in (4). Action a_{2first} is executed upon arrival of event e_2 , i.e., input place p_2 is triggered. The action has to make sure that the bindings are modified in correspondence to (6), which describe the new state (node 2 in Fig. 21):

$$a_{2first} = \{b(e_1) := a_{1after2}; b(e_2) := a_{error}\}. \quad (8)$$

The binding $b(e_t)$ need not be adjusted, since it stays the same. In this state only event e_1 can be processed, and Fig. 21 shows that it leads over node 3 and node 4 back to the stable node 0. This means that action $a_{1after2}$ has to restore the initial bindings b_0 , but also has to signal t_4 , since it is an output transition that other parts of the controller design may depend on. The implementation of $a_{1after2}$ is given as follows:

$$a_{1after2} = \{b(e_1) := a_{1first}; b(e_2) := a_{2first}; signal(t_4)\}. \quad (9)$$

Triggering input place p_1 produces event e_1 and evokes a state transition from node 0 to the quasistable state represented by node 1 of the reachability graph (Fig. 21). Equation (10) defines the necessary instructions. These include the scheduling of event e_t after 2000 time units:

$$\begin{aligned} a_{1first} &= \{b(e_1) := a_{error}; b(e_2) := a_{2after1}; \\ & b(e_t) := a_{timeout}; schedule(e_t, 2000)\}. \end{aligned} \quad (10)$$

This automatically fires event e_t after the expiration of the scheduled time. If nothing else happens meanwhile that changes the bindings, action $a_{timeout}$ that models the timeout condition will be executed (cf. path from node 1 over node 5 to node 0).

$$a_{timeout} = \{b(e_1) := a_{1first}; b(e_2) := a_{2first}; signal(t_3)\}. \quad (11)$$

The last action that has to be defined is $a_{2after1}$, which realizes the path from node 1 over 3 and 4 to node 0 of the reachability graph

$$\begin{aligned} a_{2after1} &= \{b(e_1) := a_{1first}; b(e_2) := a_{2first}; \\ & deschedule(e_t); signal(t_4)\}. \end{aligned} \quad (12)$$

The definitions (8)–(12) lead to the complete set of actions

$$A = \{a_{1first}, a_{2first}, a_{2after1}, a_{1after2}, a_{timeout}, a_{error}\}. \quad (13)$$

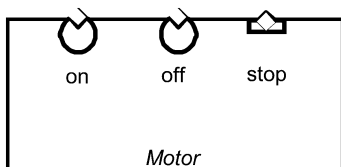


Fig. 22. Vertical interface of process component Motor.

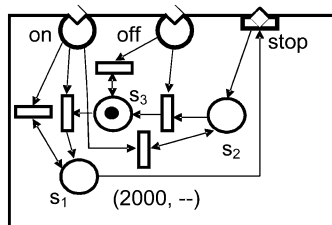


Fig. 23. Process component imitating the behavior of the “real” hardware.

```

1 public interface Motor {
2     VoidInput getOn();
3     VoidInput getOff();
4     VoidOutput getStop();
5 }

```

Fig. 24. Java interface derived from CNet interface.

C. Implementation Aspects

This kind of implicit state-space representation allows several realizations in software. One is the implementation of an event queue that is served in a one-processor, single-threaded “big loop.” This corresponds to the way traditional real-time software often has been organized. The advantage of the traditional “big loop” is a low complexity concerning determinism considerations, since the critical path is in general easy to determine. But the approach presented here also offers the possibility to let more than one execution thread in a multithreaded environment receive and process events. This is based on the binding mechanism. The function b [see (3)] can be considered a global directory of what to do when a certain event happens. And as the bound actions consist of instructions with all information needed to perform the state transitions, it does not matter which thread executes the instruction sequence. The idea of a multithreaded execution can be extended to a distributed execution on different platforms that are connected via a real-time network, e.g., RTnet [12] or a field bus like CAN or similar. One important point that has not yet been mentioned is that the selection and the execution of an action together have to be performed atomically in a multithreaded environment. This is essential, because the selection of one action changes the binding function, which again might influence the next application of it. If, e.g., the event e_2 arrives shortly after event e_1 , while the action at the previous instant associated with e_1 is still running, i.e., action $a_{1\text{first}}$, it must be assured that action $a_{2\text{after1}}$ is accessed and not $a_{2\text{first}}$. In this example event, e_1 and e_2 are strongly related to each other, so that this is necessary. A more complex component with concurrent subnets can lead to events and actions that are independent from each other and thus would not need locking or similar means applied for the implementation of atomicity. Actual work focuses on finding minimal conflict sets in order to implement atomicity with a minimum number of locks.

```

1 class MotorDriver implements Motor {
2     /* system dependent constants */
3     final int MOTOR_IO_ADDR = ...;
4     final int SWITCH_INTR_NO = ...;
5
6     private VoidInput on = new VoidInput() {
7         public void put() {
8             cnet.DirectAccess.putByte(
9                 MOTOR_IO_ADDR, 0xFF);
10        }
11        public boolean isFree() {
12            return true;
13        }
14    };
15    private VoidInput off = new VoidInput() {
16        public void put() {
17            cnet.DirectAccess.putByte(
18                MOTOR_IO_ADDR, 0x00);
19        }
20        public boolean isFree() {
21            return true;
22        }
23    };
24    private VoidOutput stop =
25        new VoidOutput() {
26        public void connect(VoidInput dest) {
27            stopTarget = dest;
28        }
29    };
30    private VoidInput stopTarget = null;
31
32    public VoidInput getOn() { return on; }
33    public VoidInput getOff() { return off; }
34    public VoidOutput getStop() {
35        return stop; }
36
37    public MotorDriver() { // Constructor
38        // install interrupt handler for
39        // position switch
40        cnet.Interrupt.install(
41            SWITCH_INTR_NO, new Runnable() {
42            public void run() {
43                if(stopTarget != null)
44                    if(stopTarget.isFree())
45                        stopTarget.put();
46            }
47        });
48    }

```

Fig. 25. Example for a driver implementation with system specific calls.

VII. DRIVER COMPONENTS

Section II introduced the different types of CNet components. The motor unit, e.g., that can be controlled by the control component from Fig. 6 must have an interface as shown in Fig. 22, i.e., it provides two parameterless inputs to switch on and off the motor, respectively, and one output that represents the position switch that indicates the arrival at the final position. The PNet implementation of the process component (see Fig. 23) tries to model the behavior of the “real” motor unit as close as possible to enable tests and the analysis of the whole system off-line by means of software tools.

In this case, the motor unit can be switched on and reaches its imaginary final position after 2000 units of time. Additional “on” events are ignored. The same applies to “off” events, when the motor is off. This behavior may be important for the analysis of the control component as described before. For the installation of the designed controller onto the hardware devices, it is necessary as explained at the beginning to map each component to a specific device and to replace process components

by drivers. These drivers have the same vertical interface and manage the real input–output communication to the plant. They incorporate knowledge about hardware addresses and device and system specific access to the peripherals. In [9], a mapping of CNet interface elements to Java language constructs has been proposed and presented, based on a software tool for the automatic generation of Java source code which has been developed by Thilo A. Grall¹. Applying this mapping to the interface definition of process component Motor (Fig. 22) leads to the code shown in Fig. 24, where the types VoidInput and VoidOutput are predefined Java interfaces representing input places and output transitions for untyped tokens, so-called *void* tokens. For more details, see [9].

Such Java interfaces serve as driver stubs, i.e., the manually written classes that realize the system and hardware specific input–output accesses assure compatibility to the design by implementing these interfaces. Fig. 25 gives an example for such driver implementation. Every device or system with a different configuration or different system calls needs its own version of this class. But the drivers are the only parts of a design that have to be adapted or updated, when hardware is changed.

VIII. SUMMARY

This paper presented CNet as a component based graphical modeling supported by a small number of textual annotations. CNet is based on the work of Wurmus [1], who conceived the idea of creating component interface from Petri net elements the way CNet does with places as inputs and transitions as outputs. The comparison of CNet with the function blocks provided by IEC 61499 has shown that CNet avoids the possibility of modeling error-prone designs resulting from data inconsistencies due to its information tagged tokens that inseparably combine events with their parameter values. The standard procedure for the reachability analysis of classical Petri nets has been extended to be applicable to CNet components producing an “extended reachability graph.” An implicit state space representation is outlined that allows the formulation of so-called “handler procedures” that enable a simple low-overhead event-discrete execution of CNet components and/or designs. Finally, driver components are explained by example showing how system and device specific implementation details are hidden by a uniform auto-generated interface.

REFERENCES

- [1] H. Wurmus, “CNet—Komponentenbasierter Entwurf verteilter Steuerungssysteme mit Petri-Netzen,” Inst. Syst. Eng., Hannover, Germany, 2002.
- [2] I. Blümke and W. M. Zuberek, “Hierarchies of place/transition refinements in Petri nets,” in *Proc. 5th IEEE Int. Conf. Emerging Technologies and Factory Automation*, Kauai, HI, Nov. 18–21, 1996, pp. 355–360.
- [3] J. P. Barros and L. Gomes, “Net model composition and modification by net operations: A Pragmatic approach,” in *Proc. 2nd IEEE Int. Conf. Industrial Informatics INDIN’04*, Berlin, Germany, Jun. 2004, pp. 309–314.

¹Entwicklung einer Software zur automatischen Codegenerierung von Java-Klassen-, -Schnittstellen- und -Methoden-Definitionen aus CNet-Sprachelementen, Bachelor’s thesis, Institut für Systems Engineering, Hannover, Germany, 2004.

- [4] K. Jensen, *Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Berlin, Germany: Springer-Verlag, 1992, vol. 1.
- [5] R. M. Shaprio, “Validation of a VLSI chip using hierarchical colored Petri nets,” in *High-Level Petri Nets—Theory and Application*, K. Jensen and G. Rozenberg, Eds. Berlin, Germany: Springer-Verlag, 1991, pp. 667–687.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*. Norwell, MA: Addison-Wesley, 1996.
- [7] “Programmable Controllers,” International Electrotechnical Commission, IEC 61131-3, 1993.
- [8] (2000) IEC 61499, Function Blocks for Industrial-Process Measurement and Control Systems, International Electrotechnical Commission. [Online]. Available: <http://www.holobloc.com>
- [9] N. Hagge and B. Wagner, “Mapping reusable control components to java language constructs,” in *Proc. 2nd IEEE Int. Conf. Industrial Informatics INDIN’04*, Berlin, Germany, Jun. 2004, pp. 108–113.
- [10] N. Hagge, B. Wagner, and B. Prause, “Internet-Access for data logging and control of a bio-technical automation platform,” in *AICHE Annu. Meeting, New Technologies for Experimentation Over the Internet*, San Francisco, CA, Nov. 17, 2003.
- [11] D. Abel, *Petri-Netze für Ingenieure—Modellbildung und Analyse diskret gesteuerter Systeme*. Berlin, Germany: Springer-Verlag, 1990.
- [12] J. Kizka, N. Hagge, P. Hohmann, and B. Wagner, “RTnet—Eine Open-Source-Lösung zur Echtzeitkommunikation über Ethernet,” in *Telematik 2003, VDI-Berichte 1785*, Siegen, Germany, Jun. 2003, pp. 55–64.



Nils Hagge received the Dipl.-Ing. degree in electrical engineering from the University of Hanover, Germany, in 2001.

In 2000, he joined the TriCore team at Infineon Technologies, San Jose, CA, for six months, working on language patterns for the automated translation of hardware descriptions from one language to another. He is now with the Real-Time Systems Group, Institute for Systems Engineering, University of Hanover. His main research interest is modeling real-time control systems and automatic code generation based on

Petri nets and real-time Java.

Mr. Hagge participated with a classmate in the young researcher’s contest “Jugend forscht” with the design of a strictly typed but machine-oriented programming language and won the first prize of the state of Lower Saxony in the field of mathematics and computer science in 1996.



Bernardo Wagner (M’97) received the M.Sc. and a Ph.D. degrees in electrical engineering from the University of Stuttgart, Germany, in 1984 and 1989, respectively.

From 1985 to 1988, he was head of the software reengineering group of GPP, Oberhaching/München, Germany. In 1991, he was appointed Professor of computer science at the University of Applied Sciences, Ulm, Germany, where he was in charge of the software laboratory. Until 1997, he worked on funded research projects and as a technical advisor of various industrial companies in the field of software testing, embedded systems, and automatic control. He has been full Professor at the University of Hanover, Germany, since March 1997, where he is in charge of the Real Time Systems Group of the Institute for Systems Engineering. He is currently the Dean of the computer science study program at the University of Hanover. Moreover, he is director of the Center for Technical Didactics at the University of Hanover, founded in 2000. He is a member of the Center for Mechatronics (MZH) and also of the research center L3S at the University of Hanover, where he works on remote services for devices, 3D-perception, real-time algorithms and system software for service robots. He gives lectures on programming, discrete control, design for real-time systems, and technical didactics. He supervises a laboratory on discrete control, a laboratory on digital hardware systems, and many student projects on autonomous robots and industrial informatics.