Component-Based Software Architectures: A Framework Based on Inheritance of Behavior

W.M.P. van der Aalst [a,c,d,1] K.M. van Hee [b,c,2]
R.A. van der Toorn [b,c,3]

[a] *Faculty of Technology Management, Department of Information and Technology, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands*

[b] *Deloitte & Touche Bakkenist, P.O. Box 23103, NL-1100 DP Amsterdam, The Netherlands*

[c] *Faculty of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands*

[d] *Department of Computer Science, University of Colorado at Boulder, Campus Box 430, Boulder, CO 80309-0430, USA*

Abstract

Software architectures shift the focus of developers from lines-of-code to coarser-grained components and their interconnection structure. Unlike fine-grained objects, these components typically encompass business functionality and need to be aware of the underlying business processes. Hence, the interface of a component should reflect relevant parts of the business process and the software architecture should emphasize the coordination among components. To shed light on these issues, we provide a framework for component-based software architectures focusing on the process perspective. The interface of a component is described in terms of Petri nets and projection inheritance is used to determine whether a component “fits”. Compositionality and substitutability are key issues for component-based development. This paper provides new results to effectively deal with these issues.

Key words: Software architectures, components, Petri nets, inheritance.

[1] E-mail: w.m.p.v.d.aalst@tm.tue.nl
[2] E-mail: kvhee@bakkenist.nl
[3] E-mail: rvdtoorn@bakkenist.nl

1 Introduction

Research in the domain of *component-based software architectures* [17,33,34] developed along two lines. On the one hand, there are contributions focussing on a formal foundation for the definition of software architectures. Examples are the many *Architecture Description Languages* (ADLs), e.g., ARMANI, Rapide, Darwin, Wright, and Aesop, that have been proposed (cf. [28]). Another example is the extension of UML based on the ROOM language [32] which allows for the specification of capsules (i.e., components), subcapsules, ports, connectors, and protocols. On the other hand, more pragmatic approaches focusing on concrete infrastructures have been developed. These approaches typically deploy *middleware* technology such as ActiveX/DCOM, CORBA, and Enterprise JavaBeans or focus on proprietary architectures such as the ones used for Enterprise Resource Planning (ERP) systems (e.g., SAP R/3 middleware). Both lines of research are characterized by a focus on the component interface and the coordination between components rather than the inner workings of components. The ultimate goal is that information systems can be assembled from large-grained components based on a thorough understanding of the business processes without detailed knowledge of the inner workings of fine-grained components (i.e., objects) [34].

In this paper, we focus on the *dynamic behavior* of components rather than the passing of data, the signature of methods, and naming issues. Since we want to reason about *consistency* of components with respect to their dynamics, we need an architectural framework which provides a formal basis for modeling and analyzing the dynamics of components. The framework presented in this paper is based on Petri nets [31]. The choice for Petri nets over other formal methods such as process algebra and state charts is primarily motivated by the availability of advanced inheritance notions and concrete inheritance-preserving transformation rules [4,10].

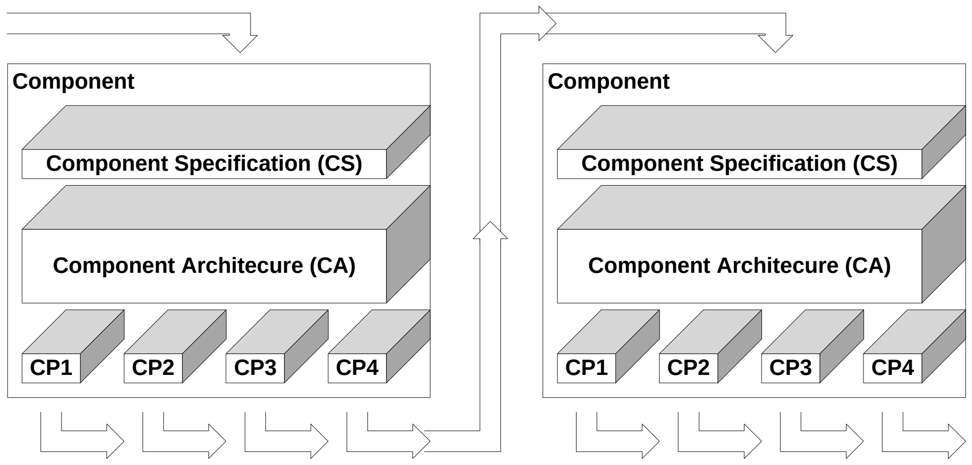

Component
Component Specification (CS)
Component Architecure (CA)
CP1
CP2
CP3
CP4
Component
Component Specification (CS)
Component Architecure (CA)
CP1
CP2
CP3
CP4

Fig. 1. A component consists of a component specification, a component architecture, and component placeholders.

Figure 1 illustrates the notion of component we will use throughout this paper. A component has a *Name* and a *Component Specification* (CS). The component specification gives the functionality *provided* by the component and is specified in terms of a particular variant of Petri nets [2] called *C-nets*. The internal structure of a component is given by a *Component Architecture* (CA). The component architecture may refer to other components by using *Component Placeholders* (CPs). Every component placeholder describes the functionality of a component used in the component architecture in terms of a C-net. A component is *closed* if it contains no other components, i.e., there are no component placeholders in its architecture. One can think of such a component as being atomic. A *System Architecture* (SA) is a set of interconnected components, i.e., component placeholders are linked to concrete components.

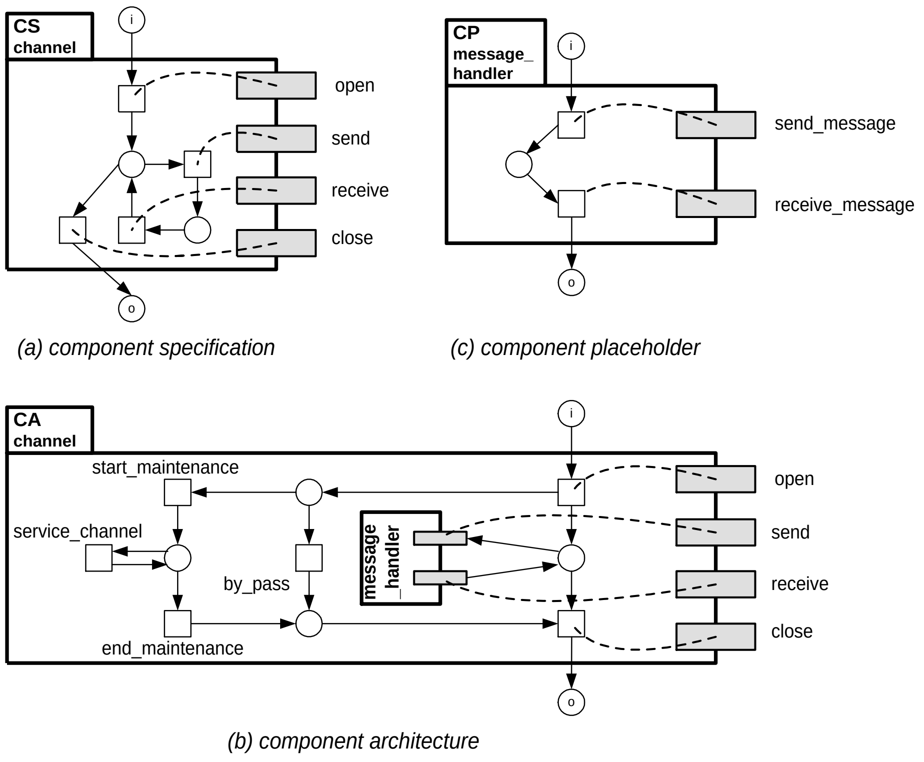
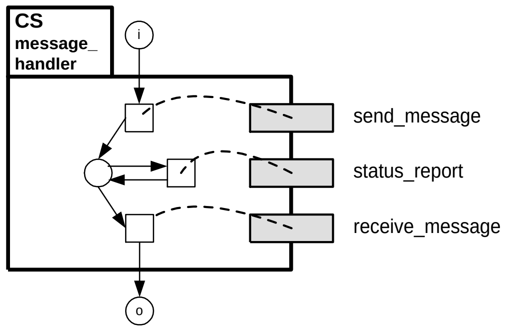
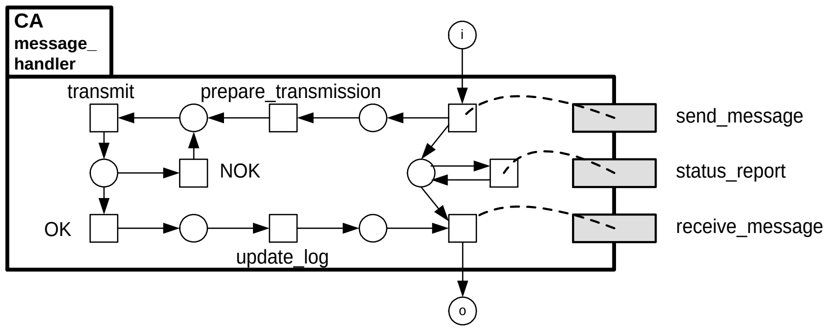


Fig. 2. The component *channel*.

Figure 2 shows a component named *channel* in terms of the framework used in this paper. The *channel* is described by (a) its specification, (b) its architecture, and (c) the specification of the only component placeholder named *message_handler*. The component specification of *channel* is given in terms of a C-net, i.e., a labeled Petri net with a uniquely identified starting point (the source place *i*) and a uniquely identified termination point (the sink place *o*). Transitions are labeled. Labels are either visible or not. Communication with the environment is via these transition labels. Figure 2(a) shows that the *channel* component has four visible labels: *open*, *send*, *receive*, and *close*. As the component specification shows, the *channel* component is activated via label *open* and deactivated via label *close*. In-between activation and deactivation, the component can send and receive messages in an alternating manner. The component specification does not describe the internal architecture of the system: It only lists the external functionality. The inner structure of the component is given by the component architecture. Figure 2(b) shows that component *channel* contains one subcomponent *message_handler* and four transitions *by_pass*, *start_maintenance*, *service_channel*, and *end_maintenance*. The transitions correspond to atomic operations which are not grouped into subcomponents. Subcomponents are specified by component placeholders. The *channel* component has only one subcomponent: *message_handler*. The corresponding component placeholder depicted in Figure 2(c) shows that the functionality of this subcomponent is straightforward: *send_message* is followed by *receive_message*. Note that the labels of the subcomponent are mapped onto labels of the *channel* component, e.g., *send_message* of *message_handler* is mapped onto *send* of *channel*. The component placeholder is not concerned with the internal structure of the subcomponent: It only specifies the minimal functionality that is expected of any component plugged into this placeholder.



(a) component specification



(b) component architecture

Fig. 3. The component *message_handler*.

Figure 3 shows another component. This component can be plugged into the placeholder of the *channel* component (i.e, *message_handler* in Figure 2(b,c)). The component shown in Figure 3 is named *message_handler* and is closed, i.e., the component does not contain any placeholders. The architecture of the *message_handler* (Figure 3) shows that for the actual transmission of messages several operations need to be performed which are not visible in the component specification, i.e., Figure 3(a), nor in the component placeholder, i.e., Figure 2(c). Although in this example the component placeholder and the component are named *message_handler*, it is not required that both bear the same name. It is required that the component realizes the functionality of the placeholder it is plugged into. Note that the component *message_handler* offers additional functionality not specified in the component placeholder of *channel*: The component *message_handler* can provide status reports but this feature is not used/required in the component *channel*.

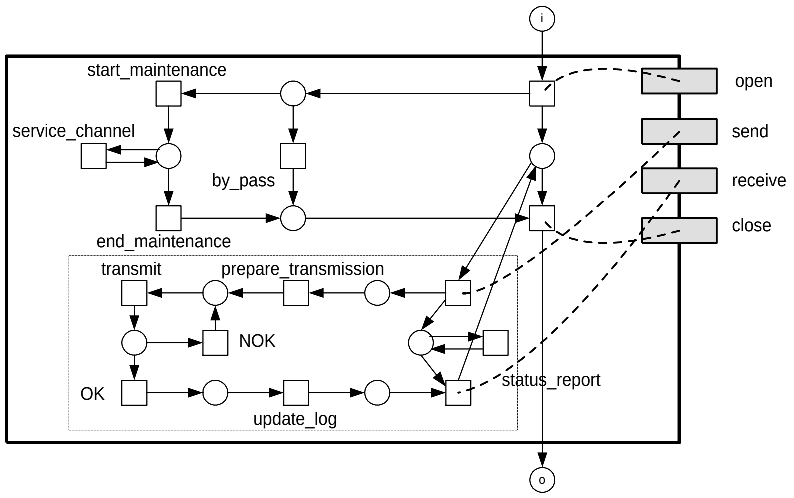


Fig. 4. The flattened system architecture composed of the *channel* and *message_handler* components.

A system architecture is composed of a set of components such that there is one top-level component and all component placeholders are mapped onto components. For example the *channel* component shown in Figure 2 where the placeholder is mapped onto the *message_handler* component shown in Figure 3, is an example of a system architecture composed of two components. The behavior of a system architecture is defined by the C-net which is obtained by recursively replacing each placeholder by a concrete component. Figure 4 shows the flattened system architecture composed of the components *channel* and *message_handler*.

The framework illustrated in Figure 1 is used to address one of the key issues of component-based software development: *consistency*. A component is consistent if, assuming the correct operation of the components that are used, its architecture actually provides the functionality specified in the component specification. A system architecture is consistent if its components are consistent and every component placeholder is mapped onto a component which actually provides the functionality specified in the component placeholder. Clearly consistency is very important in the context of component-based software development: Will a component “fit” or not? Consider for example the component *channel*. Does the architecture of *channel* shown in Figure 2(b) realize the specification shown in Figure 2(a)? Moreover, does the component *message_handler* shown in Figure 3 realize the specification of the placeholder shown in Figure 2(c)? In this paper, we address these consistency issues.

Consistency can be characterized by the term *substitutability*: Will the system operate as specified if the specification is replaced by the actual component? There are clear links between substitutability and inheritance. In earlier publications we proposed four notions of inheritance [4,10]. This paper uses the notion of *projection inheritance* to check whether a component actually provides the external behavior required. The inheritance notion is equipped with concrete inheritance-preserving design patterns and allows for modular conformance testing of the system architecture. Moreover, the replacement of one component by another is supported in two ways: (1) projection inheritance can be used to test locally whether the new component has the desired behavior, and (2) the transfer rules defined in [5] allow for automatic on-the-fly reconfiguration (i.e., migration while the component is active) by mapping the state of the old component onto the new component.

The main result of the paper is a theorem which shows that projection inheritance is compositional, i.e., if a fragment of a Petri net is replaced by another fragment which is a subclass of the original fragment, then the resulting Petri net is a subclass of the original Petri net.

Consider again the system architecture composed of the components *channel* and *message_handler*. If in Figure 2(b) the placeholder is replaced by its specification shown in Figure 2(c), then the resulting C-net is a subclass of the component specification shown in Figure 2(a) under projection inheritance. The architecture shown in Figure 3(b) is a subclass of the specification shown in Figure 3(a). The component specification shown in Figure 3(a) is also a subclass of the component placeholder shown in Figure 2(c). Based on the compositionality of projection inheritance, we can prove that from these three properties it automatically follows that the flattened system architecture shown in Figure 4 is a subclass of the specification of the top-level component shown in Figure 2(a).

The remainder of the paper is organized as follows. First, we introduce the notions this work builds upon (i.e., Petri nets, C-nets, soundness, branching bisimulation, and projection inheritance). Then, we introduce the framework for component-based software architectures followed by the main result of this paper: the proof that a consistent component architecture actually provides the external behavior it promises. To conclude, we point out some related work and discuss future extensions.

2 Preliminaries

2.1 Place/Transition nets

In this section, we define a variant of the classic Petri-net model, namely labeled Place/Transition nets. For a more elaborate introduction to Petri nets, the reader is referred to [13,30,31]. Let U be some universe of identifiers; let L be some set of *action labels*. $L_v = L \backslash \{\tau\}$ is the set of all visible labels. (The role of τ, the silent action, will be explained later.)

Definition 1 (Labeled P/T-net) *A labeled Place/Transition net is a tuple* (P, T, M, F, ℓ) *where:*

- $P \subseteq U$ *is a finite set of* places,
- $T \subseteq U$ *is a finite set of* transitions *such that* $P \cap T = \emptyset$,
- $M \subseteq L_v$ *is a finite set of* methods *such that* $M \cap (P \cup T) = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation, *and*
- $\ell : T \rightarrow M \cup \{\tau\}$ *is a* labeling function.

Each transition has a label which refers to the *method* or *operation* that is executed if the transition fires. However, if the transition bears a τ label, then no method is executed. Note that there can be many transitions with the same label, i.e., executing the same method.

Let (P, T, M, F, ℓ) be a labeled P/T-net. Elements of $P \cup T$ are referred to as *nodes*. A node $x \in P \cup T$ is called an *input node* of another node $y \in P \cup T$ if and only if there exists a directed arc from x to y; that is, if and only if xFy. Node x is called an *output node* of y if and only if there exists a directed arc from y to x. If x is a place in P, it is called an input place or an output place; if it is a transition, it is called an input or an output transition. The set of all input nodes of some node x is called the *preset* of x; its set of output nodes is called the *postset*. Two auxiliary functions $\bullet\_, \_\bullet : (P \cup T) \rightarrow \mathcal{P}(P \cup T)$ are defined that assign to each node its preset and postset, respectively. For any node $x \in P \cup T$, $\bullet x = \{y \mid yFx\}$ and $x\bullet = \{y \mid xFy\}$. Note that the preset and postset functions depend on the context, i.e., the P/T-net the function applies to. If a node is used in several nets, it is not always clear to which P/T-net the preset/postset functions refer. Therefore, we augment the preset and postset notation with the name of the net whenever confusion is possible: $\overset{N}{\bullet}x$ is the preset of node x in net N and $x\overset{N}{\bullet}$ is the postset of node x in net N.

Definition 2 (Marked, labeled P/T-net) *A* marked, *labeled P/T-net is a pair* (N, s), *where* $N = (P, T, M, F, \ell)$ *is a labeled P/T-net and where* s *is a bag over* P *denoting the marking (also called state) of the net. The set of all marked, labeled P/T-nets is denoted* $\mathcal{N}$.

For some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of occurrences of a in X, often called the cardinality of a in X. The set of all bags over A is denoted $\mathcal{B}(A)$. The empty bag, which is the function yielding 0 for any element in A, is denoted $\mathbf{0}$. For the explicit enumeration of a bag we use square brackets and superscripts to denote the cardinality of the elements. For example, $[a^2, b, c^3]$ denotes the bag with two elements a, one b, and three elements c. In this paper, we allow the use of sets as bags.

Definition 3 (Transition enabling) *Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$, where $N = (P, T, M, F, \ell)$. A transition $t \in T$ is* enabled*, denoted $(N, s)[t\rangle$, if and only if each of its input places p contains a token. That is, $(N, s)[t\rangle \Leftrightarrow \bullet t \leq s$.*

If a transition t is enabled in marking s (notation: $(N, s)[t\rangle$), then t can fire. If, in addition, t has label a (i.e., $a = \ell(t)$ is the associated method, operation, or observable action) and firing t results in marking s', then $(N, s)\ [a\rangle\ (N, s')$ is used to denote the potential firing.

Definition 4 (Firing rule) *The firing rule $\_\ [\_\rangle\ \_ \subseteq \mathcal{N} \times L \times \mathcal{N}$ is the smallest relation satisfying for any (N, s) in $\mathcal{N}$, with $N = (P, T, M, F, \ell)$, and any $t \in T$, $(N, s)[t\rangle \Rightarrow (N, s)\ [\ell(t)\rangle\ (N, s -\ \bullet t + t\bullet\ )$.*

Definition 5 (Firing sequence) *Let (N, s_0) with $N = (P, T, M, F, \ell)$ be a marked, labeled P/T-net in $\mathcal{N}$. A sequence $\sigma \in T^*$ is called a* firing sequence *of (N, s_0) if and only if $\sigma = \varepsilon$ or, for some positive natural number $n \in \mathbb{N}$, there exist markings $s_1, \ldots, s_n \in \mathcal{B}(P)$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = t_1 \ldots t_n$ and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i -\ \bullet t_{i+1} + t_{i+1}\bullet$. Sequence σ is said to be* enabled *in marking s_0, denoted $(N, s_0)[\sigma\rangle$. Firing the sequence σ results in the unique marking s, denoted $(N, s_0)\ [\sigma\rangle\ (N, s)$, where $s = s_0$ if $\sigma = \varepsilon$ and $s = s_n$ otherwise.*

Definition 6 (Reachable markings) *The set of* reachable markings *of a marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, M, F, \ell)$, denoted $[N, s\rangle$, is defined as the set $\{s' \in \mathcal{B}(P) \mid (\exists\, \sigma : \sigma \in T^* : (N, s)\ [\sigma\rangle\ (N, s'))\}$.*

Definition 7 (Connectedness) *A labeled P/T-net $N = (P, T, M, F, \ell)$ is* weakly connected*, or simply* connected*, if and only if, for every two nodes x and y in $P \cup T$, $x(F \cup F^{-1})^* y$. Net N is* strongly connected *if and only if, for every two nodes x and y in $P \cup T$, xF^*y.*

Definition 8 (Directed path) *Let (P, T, M, F, ℓ) be a labeled P/T-net. A path C from a node n_1 to a node n_k is a sequence $\langle n_1, n_2, \ldots, n_k\rangle$ such that $n_i F n_{i+1}$ for $1 \leq i \leq k - 1$. C is elementary if and only if for any two nodes n_i and n_j on C, $i \neq j\ \Rightarrow\ n_i \neq n_j$. C is non-trivial if and only if it contains at least two nodes.*

Definition 9 (Union of labeled P/T-nets) *Let $N_0 = (P_0, T_0, M_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, M_1, F_1, \ell_1)$ be two labeled P/T-nets such that $(P_0 \cup P_1) \cap (T_0 \cup T_1) = \emptyset$*

and such that, for all $t \in T_0 \cap T_1$, $\ell_0(t) = \ell_1(t)$. The union $N_0 \cup N_1$ of N_0 and N_1 is the labeled P/T-net $(P_0 \cup P_1, T_0 \cup T_1, F_0 \cup F_1, \ell_0 \cup \ell_1)$. If two P/T-nets satisfy the abovementioned two conditions, their union is said to be well defined.

Definition 10 (Boundedness) *A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ is* bounded *if and only if the set of reachable markings $[N, s\rangle$ is finite.*

Definition 11 (Safeness) *A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, M, F, \ell)$ is* safe *if and only if, for any reachable marking $s' \in [N, s\rangle$ and any place $p \in P$, $s'(p) \leq 1$.*

Definition 12 (Dead transition) *Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$. A transition $t \in T$ is* dead *in (N, s) if and only if there is no reachable marking $s' \in [N, s\rangle$ such that $(N, s')[t\rangle$.*

Definition 13 (Liveness) *A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, M, F, \ell)$ is* live *if and only if, for every reachable marking $s' \in [N, s\rangle$ and transition $t \in T$, there is a reachable marking $s'' \in [N, s'\rangle$ such that $(N, s'')[t\rangle$.*

2.2 Component nets

For the modeling of components we use labeled P/T-nets with a specific structure. We will name these nets *component nets* (C-nets).

Definition 14 (C-net) *Let $N = (P, T, M, F, \ell)$ be a labeled P/T-net. Net N is a* component net *(C-net) if and only if the following conditions are satisfied:*

(1) instance creation: P contains an input (source) place $i \in U$ such that $\bullet i = \emptyset$,
(2) instance completion: P contains an output (sink) place $o \in U$ such that $o\bullet = \emptyset$,
(3) connectedness: $\bar{N} = (P, T \cup \{\bar{t}\}, M, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, \ell \cup \{(\bar{t}, \tau)\})$ is strongly connected, and
(4) visibility: for any $t \in T$ such that $t \in (i\bullet \cup \bullet o)$: $\ell(t) \in L_v$.

Note that the connectedness requirement implies that there is one unique source and one unique sink place. For the readers familiar with the work presented in [1–3]: C-nets are WF-nets with the additional requirement that the start transitions $i\bullet$ and end transitions $\bullet o$ have a non-τ label. Figures 2(a), 2(c), and 3(a) show examples of C-nets. The structure of a C-net allows us to define the following functions.

Definition 15 ($\underline{source}, \underline{sink}, \underline{start}, \underline{stop}, \underline{strip}$) *Let $N = (P, T, M, F, \ell)$ be a C-net.*

(1) $\underline{source}(N)$ is the (unique) input place $i \in P$ such that $\bullet i = \emptyset$,

(2) $\underline{sink}(N)$ *is the (unique) output place* $o \in P$ *such that* $o\bullet = \emptyset$*,*
(3) $\underline{start}(N) = \{t \in T \mid i \in \bullet t\}$ *is the set of start transitions,*
(4) $\underline{stop}(N) = \{t \in T \mid o \in t\bullet\}$ *is the set of stop transitions, and*
(5) $\underline{strip}(N) = (P', T, M, F \cap ((P' \times T) \cup (T \times P')), \ell)$ *with* $P' = P \setminus \{\underline{source}(N), \underline{sink}(N)\}$ *is the C-net without source and sink place.*

Definition 14 only gives a static characterization of a C-net. Components will have a life-cycle which satisfies the following requirements.

Definition 16 (Soundness) *A C-net N with $\underline{source}(N) = i$ and $\underline{sink}(N) = o$ is said to be* sound *if and only if the following conditions are satisfied:* [4]

(1) safeness: $(N, [i])$ *is safe,*
(2) proper completion: for any reachable marking $s \in [N, [i]\rangle$*,* $o \in s$ *implies* $s = [o]$*,*
(3) completion option: for any reachable marking $s \in [N, [i]\rangle$*,* $[o] \in [N, s\rangle$*, and*
(4) dead transitions: $(N, [i])$ *contains no dead transitions.*

The set of all sound C-nets is denoted $\mathcal{C}$. The first requirement states that a sound C-net is safe. The second requirement states that the moment a token is put in place o all the other places should be empty, which corresponds to the termination of a component without leaving dangling references. The third requirement states that starting from the initial marking $[i]$, i.e., activation of the component, it is always possible to reach the marking with one token in place o, which means that it is always feasible to terminate successfully. The last requirement, which states that there are no dead transitions, corresponds to the requirement that for each transition there is an firing sequence activating this transition. Note that each of the C-nets shown in Figures 2(a), 2(c), and 3(a) is sound.

The following theorem shows that soundness can be expressed in terms of two well-known properties: liveness and safeness.

Theorem 17 (Characterization of soundness) *Let $N = (P, T, M, F, \ell)$ be a C-net and $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, \ell \cup \{(\bar{t}, \tau)\})$ the short-circuited version of N. N is sound if and only if $(\bar{N}, [i])$ is live and safe.*

PROOF. The proof is similar to the proof of Theorem 11 in [1]. The only difference is that in this paper a stronger notion of soundness is used, which implies safeness rather than boundedness of the short-circuited net. □

The fact that soundness coincides with standard properties such as liveness and safeness allows us to use existing tools and techniques to verify soundness of a

[4] Note that $[i]$ and $[o]$ are *bags* containing the input respectively output place of N.

given C-net.

The following lemma shows that start/stop transitions share a unique source/sink place.

Lemma 18 *Let $N = (P, T, M, F, \ell)$ be a sound C-net, i.e., $N \in \mathcal{C}$. For any $t \in T$, (i) if $i = \underline{source}(N)$ and $t \in \underline{start}(N)$, then $\bullet t = \{i\}$, and (ii) if $o = \underline{sink}(N)$ and $t \in \underline{stop}(N)$, then $t\bullet = \{o\}$.*

PROOF. See [3]. $\square$

The alphabet operator α is a function yielding the set of visible labels of all transitions of the net that are not dead.

Definition 19 (Alphabet operator α) *Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$, with $N = (P, T, M, F, \ell)$. $\alpha : \mathcal{N} \to \mathcal{P}(L_v)$ is a function such that $\alpha(N, s) = \{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau \wedge t \text{ is not dead}\}$.*

Since sound C-nets do not contain dead transitions, $\alpha(N, [i])$ equals $\{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau\}$, which is denoted by $\alpha(N)$.

2.3 Branching bisimilarity

To formalize projection inheritance, we need to formalize a notion of equivalence. In this paper, we use *branching bisimilarity* [18] as the standard equivalence relation on marked, labeled P/T-nets in $\mathcal{N}$.

The notion of a *silent action* is pivotal to the definition of branching bisimilarity. Silent actions are actions (i.e., transition firings) that cannot be observed. Silent actions are denoted with the label τ, i.e., only transitions in a P/T-net with a label different from τ are observable. Note that we assume that τ is an element of L. The τ-labeled transitions are used to distinguish between external, or observable, and internal, or silent, behavior. A single label is sufficient, since all internal actions are equal in the sense that they do not have any visible effects.

In the context of components, we want to distinguish *successful termination* from *deadlock*. A *termination predicate* defines in what states a marked P/T-net can terminate successfully. If a marked, labeled P/T-net is in a state where it cannot perform any actions or terminate successfully, then it is said to be in a *deadlock*. Based on the notion of soundness, successful termination corresponds to the state with one token in the sink place.

Definition 20 *The class of marked, labeled P/T-nets $\mathcal{N}$ is equipped with the following termination predicate:* $\downarrow = \{(N, [o]) \mid N$ *is a C-net* $\wedge\, o = \underline{sink}(N)\}$.

To define branching bisimilarity, two auxiliary definitions are needed: (1) a relation expressing that a marked, labeled P/T-net can evolve into another marked, labeled P/T-net by executing a sequence of zero or more τ actions; (2) a predicate expressing that a marked, labeled P/T-net can terminate by performing zero or more τ actions.

Definition 21 *The relation* $\_ \Longrightarrow \_ \subseteq \mathcal{N} \times \mathcal{N}$ *is defined as the smallest relation satisfying, for any* $p, p', p'' \in \mathcal{N}$, $p \Longrightarrow p$ *and* $(p \Longrightarrow p' \wedge p' \,[\tau\rangle\, p'') \;\Rightarrow\; p \Longrightarrow p''$.

Definition 22 *The predicate* $\Downarrow \_ \subseteq \mathcal{N}$ *is defined as the smallest set of marked, labeled P/T-nets satisfying, for any* $p, p' \in \mathcal{N}$, $\downarrow p \Rightarrow \Downarrow p$ *and* $(\Downarrow p \wedge p' \,[\tau\rangle\, p) \;\Rightarrow\; \Downarrow p'$.

Let, for any two marked, labeled P/T-nets $p, p' \in \mathcal{N}$ and action $\alpha \in L$, $p \,[(\alpha)\rangle\, p'$ be an abbreviation of the predicate $(\alpha = \tau \wedge p = p') \vee p \,[\alpha\rangle\, p'$. Thus, $p \,[(\tau)\rangle\, p'$ means that zero τ actions are performed, when the first disjunct of the predicate is satisfied, or that one τ action is performed, when the second disjunct is satisfied. For any observable action $a \in L\backslash\{\tau\}$, the first disjunct of the predicate can never be satisfied. Hence, $p \,[(a)\rangle\, p'$ is simply equal to $p \,[a\rangle\, p'$, meaning that a single a action is performed.

Definition 23 (Branching bisimilarity) *A binary relation* $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ *is called a* branching bisimulation *if and only if, for any* $p, p', q, q' \in \mathcal{N}$ *and* $\alpha \in L$,

(1) $p\mathcal{R}q \wedge p \,[\alpha\rangle\, p' \Rightarrow$
$(\exists\, q', q'' : q', q'' \in \mathcal{N} : q \Longrightarrow q'' \wedge q'' \,[(\alpha)\rangle\, q' \wedge p\mathcal{R}q'' \wedge p'\mathcal{R}q')$,
(2) $p\mathcal{R}q \wedge q \,[\alpha\rangle\, q' \Rightarrow$
$(\exists\, p', p'' : p', p'' \in \mathcal{N} : p \Longrightarrow p'' \wedge p'' \,[(\alpha)\rangle\, p' \wedge p''\mathcal{R}q \wedge p'\mathcal{R}q')$, *and*
(3) $p\mathcal{R}q \Rightarrow (\downarrow p \Rightarrow \Downarrow q \wedge \downarrow q \Rightarrow \Downarrow p)$.

Two marked, labeled P/T-nets are called branching bisimilar, *denoted* $p \sim_b q$, *if and only if there exists a branching bisimulation* $\mathcal{R}$ *such that* $p\mathcal{R}q$.

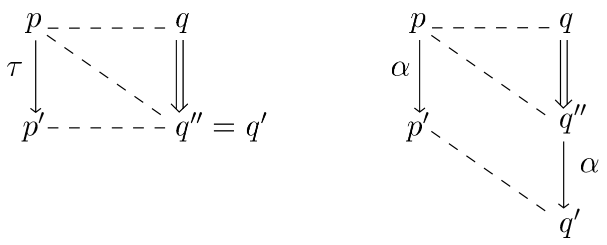

$$\begin{tikzcd}
p \arrow[r, dash, dashed] \arrow[d, "\tau"'] \arrow[dr, dash, dashed] & q \arrow[d, Rightarrow] \\
p' \arrow[r, dash, dashed] & q'' = q'
\end{tikzcd}
\qquad
\begin{tikzcd}
p \arrow[r, dash, dashed] \arrow[d, "\alpha"'] \arrow[dr, dash, dashed] & q \arrow[d, Rightarrow] \\
p' \arrow[ddr, dash, dashed] & q'' \arrow[d, "\alpha"] \\
 & q'
\end{tikzcd}$$

Fig. 5. The essence of a branching bisimulation.

Figure 5 shows the essence of a branching bisimulation. The firing rule is depicted by arrows. The dashed lines represent a branching bisimulation. A marked, labeled P/T-net must be able to simulate any action of an equivalent marked, labeled P/T-net after performing any number of silent actions, except for a silent action which it may or may not simulate. The third property in Definition 23 guarantees that related marked, labeled P/T-nets always have the same termination options.

Branching bisimilarity is an equivalence relation on $\mathcal{N}$, i.e., $\sim_b$ is reflexive, symmetric, and transitive. See [10] for more details and pointers to other notions of branching bisimilarity.

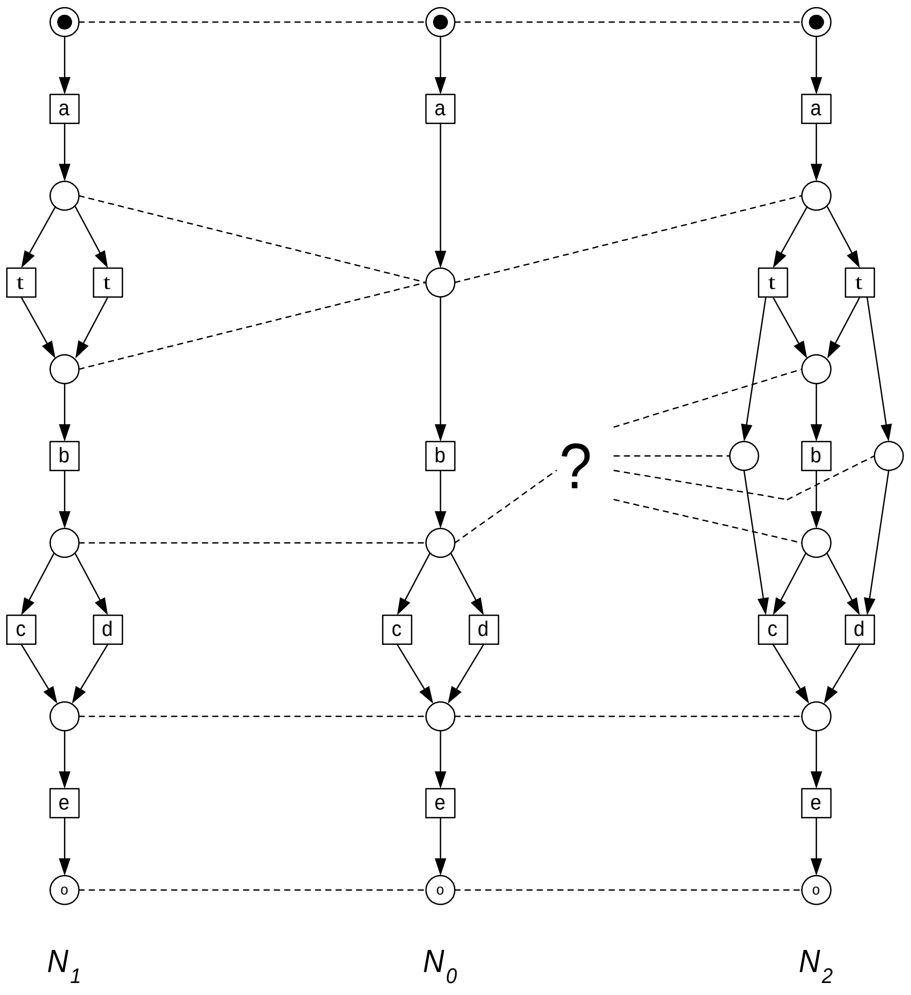


Fig. 6. Three marked C-nets: the first two are branching bisimilar and the third one is not branching bisimilar to the other two.

To illustrate the relevance of branching bisimilarity as an equivalence notion we use the three marked C-nets shown in Figure 6. Each of the nets has the following visible behavior: either the trace $abce$ is realized or trace $abde$ is realized. Therefore, it is interesting to investigate whether the three marked C-nets are branching bisimilar. $(N_0, [i])$ and $(N_1, [i])$ are branching bisimilar. However, $(N_0, [i])$ and $(N_2, [i])$ are not, i.e., although they are trace equivalent $(N_0, [i]) \not\sim_b (N_2, [i])$! The reason is that in N_0 the moment of choice between c and d is made *after* the execution of b while in N_2 the choice is made *before* the execution of b. This distinction is vital when dealing with components. Assume that b corresponds to sending a request to a component and that c is executed in case of a positive response and that d is executed in case of a negative response. In N_0 the C-net can handle both a positive response (c) and a negative response (d) after sending the request (b). However, in N_2 the C-net can handle either the positive or the negative response, i.e., the choice between c and d is made before the execution of b. Clearly, the latter C-net is not acceptable, since it assumes that before sending the request the answer of the supplier is already known. This simple example shows that straightforward notions of equivalence such as trace equivalence (after abstraction of internal steps) are not selective enough for the problems addressed in this paper. Therefore, we use the more refined notion of branching bisimilarity.

Definition 24 (Behavioral equivalence of C-nets) *For any two C-nets N_0 and N_1 in $\mathcal{C}$, $N_0 \cong N_1$ if and only if $(N_0, [i]) \sim_b (N_1, [i])$.*

Consider the three nets shown in Figure 6: $N_0 \cong N_1$, $N_0 \not\cong N_2$, and $N_1 \not\cong N_2$.

2.4 Inheritance

In [4,5,10] four notions of inheritance have been identified. Unlike most other notions of inheritance, these notions focus on the dynamics rather than data and/or signatures of methods. These inheritance notions address the usual aspects: (1) *substitutability* (Can the superclass be replaced by the subclass without breaking the system?), (2) *subclassing* (implementation inheritance: Can the subclass use the implementation of the superclass?), and (3) *subtyping* (interface inheritance: Can the subclass use or conform to the interface of the superclass?). The four inheritance notions are inspired by a mixture of these three aspects.

In this paper, we restrict ourselves to one of the four inheritance notions: *projection inheritance*. In the future we hope to extend our component framework with other notions of inheritance (cf. Section 7). The basic idea of projection inheritance can be characterized as follows.

> *If it is not possible to distinguish the behaviors of x and y when arbitrary methods of x are executed, but when only the effects of methods that are also present in y are considered, then x is a subclass of y.*

For projection inheritance, all new methods (i.e., methods added in the subclass) are hidden. Therefore, we introduce the abstraction operator τ_I that can be used to hide methods.

Definition 25 (Abstraction) *Let $N = (P, T, M, F, \ell_0)$ be a labeled P/T-net. For any $I \subseteq L_v$, the abstraction operator τ_I is a function that renames all transition labels in I to the silent action τ. Formally, $\tau_I(N) = (P, T, M, F, \ell_1)$ such that, for* *any* $t \in T$, $\ell_0(t) \in I$ *implies* $\ell_1(t) = \tau$ *and* $\ell_0(t) \notin I$ *implies* $\ell_1(t) = \ell_0(t)$.

The definition of projection inheritance is straightforward, given the abstraction operator and branching bisimilarity as an equivalence notion.

Definition 26 (Inheritance) *For any two sound C-nets* N_0 *and* N_1 *in* $\mathcal{C}$, N_1 *is a subclass of* N_0 *under projection inheritance, denoted* $N_1 \leq_{pj} N_0$, *if and only if there is an* $I \subseteq L_v$ *such that* $(\tau_I(N_1), [i]) \sim_b (N_0, [i])$.

It is easy to show that $\leq_{pj}$ is a partial order, i.e., $\leq_{pj}$ is reflexive, anti-symmetric, and transitive [10].

Proposition 27 *Assuming* $\cong$, *as defined in Definition 24, as the equivalence on sound C-nets* $\leq_{pj}$ *is a partial order.*

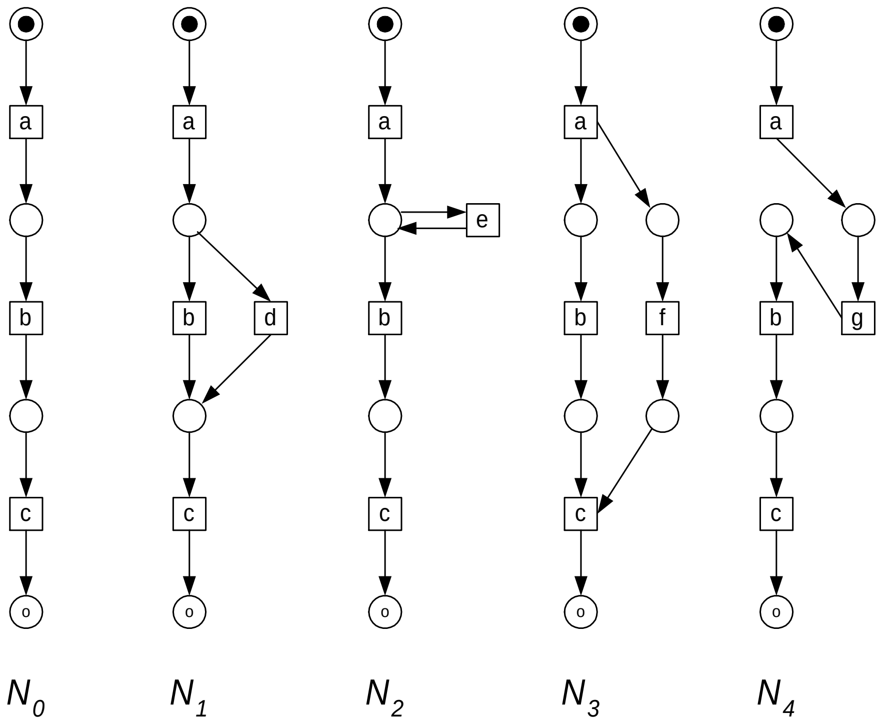


Fig. 7. N_2, N_3, and N_4 are subclasses of N_0 under projection inheritance.

Let us consider the five C-nets shown in Figure 7 to illustrate the notion of projection inheritance. N_1 is *not* a subclass of N_0 because hiding of the new task d results in a potential trace where a is followed by c without executing b, i.e., the C-net where d is renamed to τ is not branching bisimilar. N_2 is a subclass of N_0 because hiding e in N_2 results in a behavior equivalent to the behavior of N_0, i.e., the addition of e only postpones the execution of b and does not allow for a bypass such as the one in N_1. N_3 is also a subclass of N_0: Hiding the parallel branch containing f yields the original behavior. Finally, N_4 is also a subclass of N_0.

Based on the notion of projection inheritance we have defined three inheritance-preserving transformation rules. These rules correspond to design patterns when extending a superclass to incorporate new behavior: (1) adding a loop, (2) inserting methods in-between existing methods, and (3) putting new methods in parallel with existing methods. Without detailed proofs we summarize some of the results given in [4,5,10].

Theorem 28 (Projection-inheritance-preserving transformation rule PPS) *Let $N_0 = (P_0, T_0, M_0, F_0, \ell_0)$ be a sound C-net in $\mathcal{C}$. If $N = (P, T, M, F, \ell)$ is a labeled P/T-net with place $p \in P$ such that*

(1) $p \notin \{i, o\}$, $P_0 \cap P = \{p\}$, $T_0 \cap T = \emptyset$,
(2) $(\forall\, t : t \in T : \ell(t) \notin \alpha(N_0))$,
(3) $(\forall\, t : t \in T \wedge p \in \bullet t : \ell(t) \neq \tau)$,
(4) $(N, [p])$ *is live and safe, and*
(5) $N_1 = N_0 \cup N$ *is well defined,*

then N_1 is a sound C-net in $\mathcal{C}$ such that $N_1 \leq_{pj} N_0$.

PROOF. N is added to N_0 such that it forms a subclass under projection inheritance. (In fact, it is a subclass under all four notions of inheritance identified in [4,5,10].) It is straightforward to prove that this is the case. The added net N forms an arbitrary complex extension which can consume tokens from place p as long as it is guaranteed that eventually every token is returned. The labels of the transitions in N should not appear in N_0. This way it is possible to abstract from them and it is possible to construct a branching bisimulation. A detailed proof can be found in [5]. $\Box$

Note that PPS can be used to construct the subclass N_2 in Figure 7 from the C-net N_0 shown in the same figure.

Theorem 29 (Projection-inheritance-preserving transformation rule PJS) *Let $N_0 = (P_0, T_0, M_0, F_0, \ell_0)$ be a sound C-net in $\mathcal{C}$. If $N = (P, T, M, F, \ell)$ is a labeled P/T-net with place $p \in P$ and transition $t_p \in T$ such that*

(1) $p \notin \{i, o\}$, $P_0 \cap P = \{p\}$, $T_0 \cap T = \{t_p\}$, $(t_p, p) \in F_0$, *and* $\overset{N}{\bullet} t_p = \{p\}$,
(2) $(\forall\, t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0))$,
(3) $(N, [p])$ *is live and safe, and*
(4) $N_1 = (P_0, T_0, M_0, F_0 \backslash \{(t_p, p)\}, \ell_0) \cup (P, T, M, F \backslash \{(p, t_p)\}, \ell)$ *is well defined,*

then N_1 is a sound C-net in $\mathcal{C}$ such that $N_1 \leq_{pj} N_0$.

PROOF. In the C-net N_0 an arc connecting transition t_p and place p is replaced by a P/T-net N. In the resulting C-net N_1, transition t_p produces tokens for places in

N instead of p. However, the properties of N guarantee that eventually every firing of transition t_p is followed by the production of a single token for p. Moreover, the moment N marks p all other places in N are empty and the labels of the transitions in N do not appear in N_0. Therefore, it is possible to abstract from the transitions in N and construct a branching bisimulation (cf. [5]) and thus it is shown that N_1 is a subclass of N_0 under projection inheritance. $\Box$

Transformation rule PJS can be used to construct N_4 from N_0 in Figure 7.

Theorem 30 (Projection-inheritance-preserving transformation rule $PJ3S$) *Let $N_0 = (P_0, T_0, M_0, F_0, \ell_0)$ be a sound C-net in $\mathcal{C}$. Let $N = (P, T, M, F, \ell)$ be a labeled P/T-net. Assume that $q \in U$ is a fresh identifier not appearing in $P_0 \cup T_0 \cup P \cup T$. If N contains a place $p \in P$ and transitions $t_i, t_o \in T$ such that*

(1) $\overset{N}{\bullet} p = \{t_o\}$, $p \overset{N}{\bullet} = \{t_i\}$,
(2) $P_0 \cap P = \emptyset$, $T_0 \cap T = \{t_i, t_o\}$,
(3) $(\forall\, t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0))$,
(4) $(N, [p])$ is live and safe,
(5) $N_1 = N_0 \cup (P \backslash \{p\}, T, F \backslash \{(p, t_i), (t_o, p)\}, \ell)$ is well defined,
(6) q is implicit in $(N_0^q, [i])$ with $N_0^q = (P_0 \cup \{q\}, T_0, F_0 \cup \{(t_i, q), (q, t_o)\}, \ell_0)$, and
(7) N_0^q is a sound C-net,

then N_1 is a sound C-net in $\mathcal{C}$ such that $N_1 \leq_{pj} N_0$.

PROOF. C-net N_0 contains two transitions t_i and t_o such that every firing of t_i is followed by precisely one firing of t_o. Given this requirement it is possible to add a P/T-net N which is executed in parallel. N is activated by t_i (i.e., places in N become marked) and is required to mark the additional input places of t_o after every activation. This implies that t_o is not constrained by the added part: It is still guaranteed that (eventually) every firing of t_i is followed by precisely one firing of t_o. Moreover, no tokens are left in N because of the requirement that $(N, [p])$ is live and safe. Note that places p and q have been added for technical reasons and do not appear in the resulting net N_1. The transitions in N have labels not appearing in N_0. Therefore, it is possible to abstract from N and construct a branching bisimulation. A detailed proof can be found in [5]. Similar proofs for theorems 28, 29, and 30 (where the safety requirements are replaced by free-choice requirements) can be found in [4,10]. $\Box$

Transformation rule $PJ3S$ can be used to construct subclass N_3 from superclass N_0 in Figure 7.

Rule PPS can be used to insert a loop or iteration at any point in the process, provided that the added part always returns to the initial state. Rule PJS can be used to insert new methods by replacing a connection between a transition and a place by an arbitrary complex subnet. Rule $PJ3S$ can be used to add parallel behavior, i.e., new methods which are exectuted in parallel with existing methods. The inheritance-preserving transformation rules distinguish the work presented in [4,5,10] from earlier work on inheritance. The rules correspond to design constructs that are often used in practice, namely iteration, sequential composition, and parallel composition. If a designer sticks to these rules, inheritance is guaranteed!

3 Framework

In this section we formalize the concepts introduced in Section 1. As illustrated by Figure 1, a *component* consists of a *component specification* (CS) and a *component architecture* (CA), and the component architecture may contain a number of *component placeholders* (CPs).

Definition 31 (Component) *A* component c *is a tuple* (CS, CA) *where:*

(1) $CS = (P^S, T^S, M^S, F^S, \ell^S)$ *is a sound C-net called the* component specification *of* c*, and*
(2) $CA = (P^A, T^A, C^A, F^A, \ell^A)$ *is the* component architecture *of* c *such that:*
  (a) $P^A \subseteq U$ *are the places in the component architecture,*
  (b) $T^A \subseteq U$ *are the transitions in the component architecture,*
  (c) C^A *is a set of* component placeholders *such that every* $cp \in C^A$ *is a component specification, i.e.,* $cp = (P^{SA}_{cp}, T^{SA}_{cp}, M^{SA}_{cp}, F^{SA}_{cp}, \ell^{SA}_{cp})$ *is a sound C-net,*
  (d) $B = \{(cp, l) \in C^A \times L_v \mid l \in M^{SA}_{cp}\}$ *is the set of* bindings,
  (e) $F^A \subseteq (P^A \times (T^A \cup B)) \cup ((T^A \cup B) \times P^A)$ *is called the* component flow relation, *and*
  (f) $\ell^A : T^A \cup B \rightarrow M^S \cup \{\tau\}$ *is the* component labeling function.

The component specification defines the interface of a component in terms of a C-net. The purpose of the component architecture is to actually realize/implement this specification, i.e., the architecture is typically much more detailed and may contain other components. For closed components $C^A = \emptyset$. For non-closed components the architecture contains a set of placeholders C^A. The placeholders are used for *plugging in* other components. Closed components are atomic in sense that it is not possible to plug in subcomponents. Each placeholder specifies the required interface of the component to be plugged in. There are two types of arcs in the architecture: (1) normal arcs (i.e., arcs between places and transitions) and (2) subcomponent arcs which connect places in the architecture to methods *inside* the components plugged into the component placeholders. To address methods inside subcomponents, a set of bindings B is introduced. Note that ℓ^A can be used to map methods inside the components plugged into the component placeholders onto methods used in the component specification, i.e., each method associated to a component placeholder is mapped onto either τ or a visible method in M^S. Moreover, ℓ^A also maps ordinary transitions in the architecture onto a label in $M^S \cup \{\tau\}$. The methods and transitions that are mapped onto τ by ℓ^A are not visible from outside the component.

Both Figure 2 and Figure 3 show examples of components. The component *message_handler* is closed; the component *channel* is not.

Figure 8 shows another example of a component. This component represents a very simple coffee machine which accepts coins and either returns coins or serves coffee. The component specification (*CS coffee_machine*) shows that after activating the machine (method *switch_on*) a coin can be inserted (method *insert_coin*). After an *internal* choice (i.e., two τ-labeled transitions sharing one input place) either method *reject_coin* or method *serve_coffee* is enabled. After executing one of these two methods the machine returns to a state where it accepts a new coin. In parallel the machine can be deactivated using the method *switch_off*. Since the machine can be busy serving coffee, there is another method (*switched_off*) which corresponds to the actual switch-off operation.

The architecture of the component *coffee_machine* is described by the remaining three diagrams in Figure 8. The two smaller diagrams correspond to component placeholders. The larger diagram in the middle describes the overall architecture of the component and refers to the two component placeholders. The component placeholder *coin_handler* takes care of accepting and rejecting coins. The component placeholder *brewing_facility* takes care of the actual brewing and serving of coffee. Note that at the architectural level one can see the interaction between components inside the machine. Both subcomponents are activated/deactivated when the machine is switched on/off. After a coin is inserted the *coin_handler* sends a request to the *brewing_facility*. The *brewing_facility* either acknowledges the request (OK) and serves coffee or sends a notification to the *coin_handler* (NOK) resulting in the returning of the coin inserted. Note that external methods (i.e., the methods offered in the component specification) are linked to concrete transitions in the architectural model or are mapped onto internal methods provided by component placeholders. Also note that places in the component architecture are connected to concrete transitions or methods provided by component placesholders, e.g., place *OK* is connected to method *OK!* of the component placeholder *brewing_facility* and method *OK?* of the component placeholder *coin_handler*.

Assumption *In the remainder we assume that there are no name clashes, i.e., all component specifications, placeholders, and component architectures use different identifiers for places and transitions. The only identifiers shared among component*

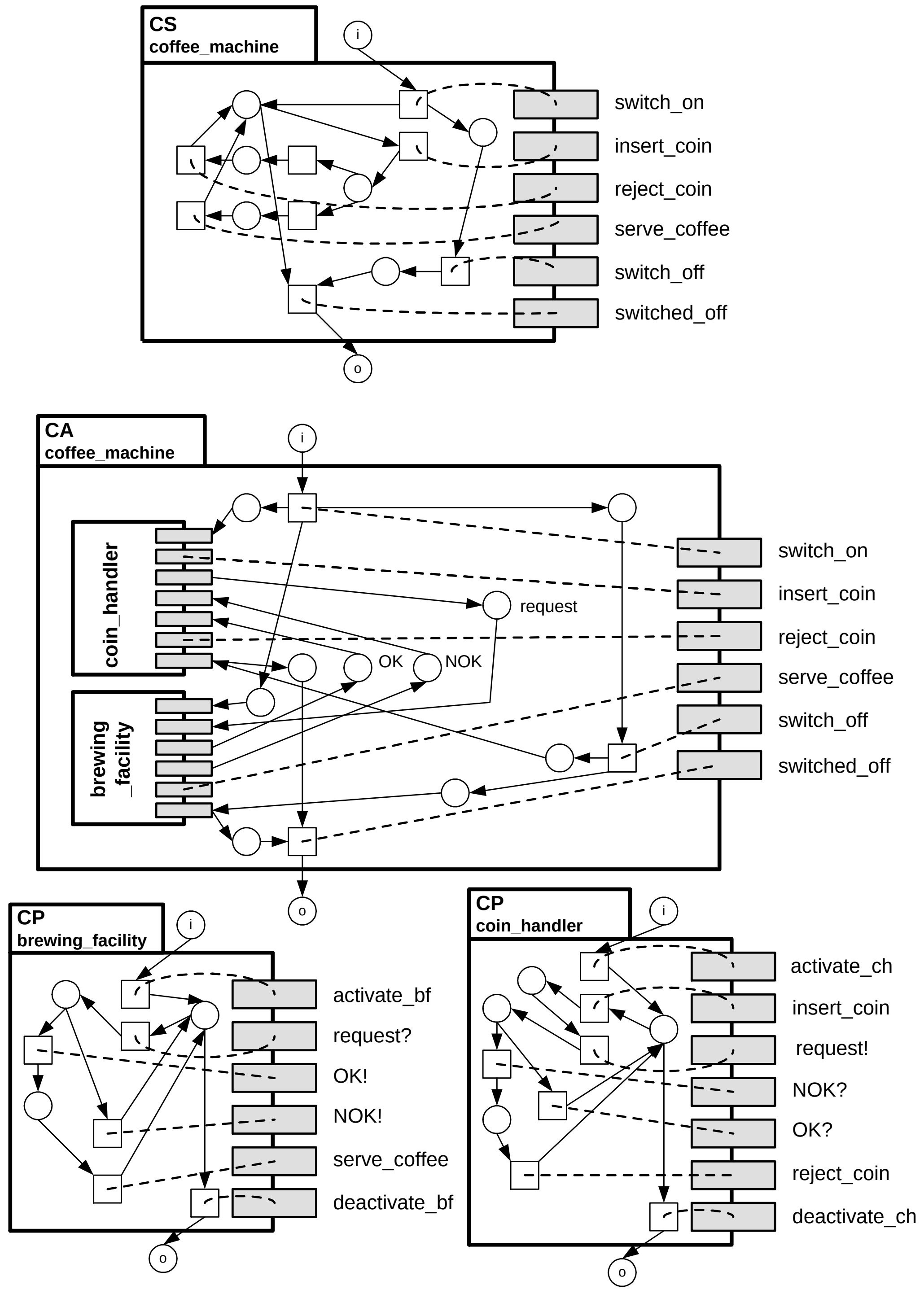


Fig. 8. The component *coffee_machine*.

specifications, placeholders, and component architectures are the action labels.

The architecture of a component should provide the functionality promised in its specification. Therefore, we define the function $\underline{cflat}$ which allows us to define *component consistency*.

Definition 32 (Flattened component) *Let $CA = (P^A, T^A, C^A, F^A, \ell^A)$ be a component architecture such that for any $cp \in C^A$: $\underline{strip}(cp) = (P^{SA}_{cp}, T^{SA}_{cp}, M^{SA}_{cp}, F^{SA}_{cp}, \ell^{SA}_{cp})$ is the stripped component specification. The corresponding flattened architecture is the labeled P/T net $\underline{cflat}(CA) = (P, T, M, F, \ell)$ with:*

(1) $P = P^A \cup (\bigcup_{cp \in C^A} P^{SA}_{cp})$,
(2) $T = T^A \cup (\bigcup_{cp \in C^A} T^{SA}_{cp})$,
(3) $F = (F^A \cap ((P^A \times T^A) \cup (T^A \times P^A))) \cup (\bigcup_{cp \in C^A} F^{SA}_{cp} \cup \{(p,t) \in P^A \times T^{SA}_{cp} \mid (p, (cp, \ell^{SA}_{cp}(t))) \in F^A\} \cup \{(t,p) \in T^{SA}_{cp} \times P^A \mid ((cp, \ell^{SA}_{cp}(t)), p) \in F^A\})$,
(4) $dom(\ell) = T$, *for any* $t \in T^A$: $\ell(t) = \ell^A(t)$, *and for any* $cp \in C^A$ *and* $t \in T^{SA}_{cp}$: $\ell(t) = \ell^A(cp, \ell^{SA}_{cp}(t))$, *and*
(5) $M = rng(\ell) \backslash \{\tau\}$.

Figure 9 shows the flattened architecture of the component *channel* shown in Figure 2: The component placeholder *message_handler* is replaced by the closed component *message_handler* shown in Figure 3.

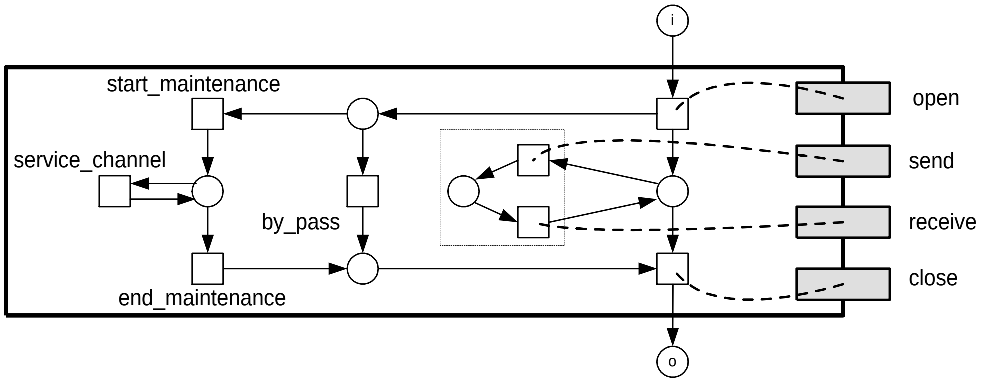


Fig. 9. The flattened component *channel*.

Components are single-threaded in the following sense: There may be parallel behavior inside the component, but the component itself is only instantiated once. Therefore, we consider the phenomenon called *multiple activation* an anomaly. To explain this anomaly we introduce the term *activation*. A component is activated if at least one of the places in the component is marked (except the source and sink place). Note that a component becomes activated after one of the start transitions fires. A component becomes deactivated if each of its internal places is empty after one of the stop transitions fires. Ideally, every activation is followed by a deactivation. The soundness property is defined for C-nets which are activated only once. Therefore, all kinds of undesired side effects can occur if a component is activated for the second time without being deactivated first. To formulate the requirement that there is no multiple activation, we define the notion of *activation safeness*.

Definition 33 (Activation safeness) *Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$,* *where $N = (P, T, M, F, \ell)$. A subset of places $P' \subseteq P$ is* activation safe *in (N, s) if and only if for any reachable state $s' \in [N, s\rangle$, any transition $t \in \bullet P' \backslash P' \bullet$, and any place $p \in P'$: $(N, s')[t\rangle$ implies $s'(p) = 0$.*

A set of places P' is activation safe if all transitions producing tokens for P' but not consuming tokens from P' are not enabled as long as there are tokens in P'.

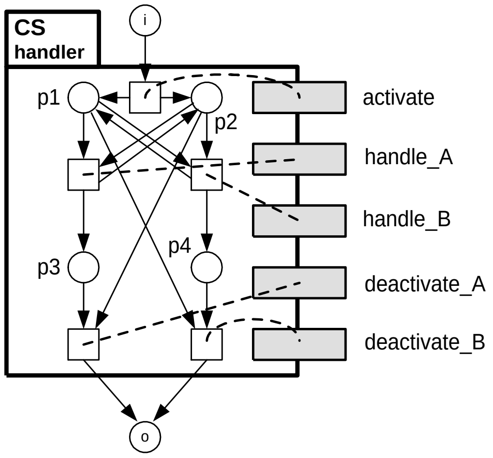


Fig. 10. The specification of a component which deadlocks after being activated twice.

To illustrate the relevance of activation safeness we use the component specification shown in Figure 10. The component specification is sound. However, if the component is activated twice, it can deadlock. Suppose that start transition *activate* is executed twice. Both *p1* and *p2* will contain two tokens. Suppose that *handle_A* and *handle_B* are executed once. In the resulting state places *p1*, *p2*, *p3*, and *p4* are marked. Suppose that *handle_A* is executed again. In the resulting state only stop transition *deactivate_A* is enabled. Firing *deactivate_A* results in the state marking *p3* and *p4*. In this state none of the transitions is enabled, i.e., the component gets stuck in a state where it is not possible to terminate properly. The deadlock is a result of the fact that a component which is activated multiple times exhibits behavior which is not considered when checking for soundness.

To avoid deadlocks such as the one illustrated using Figure 10, the architecture of each component should be such that each of its subcomponents are activation safe. Using Definition 33, we can formulate the notion of *consistency*. In a consistent component each subcomponent should be activation safe. Moreover, the flattened component should be sound and a subclass of the component specification.

Definition 34 (Consistent) *Let (CS, CA) be a component with $CS = (P^S, T^S, M^S, F^S, \ell^S)$, $CA = (P^A, T^A, C^A, F^A, \ell^A)$, and for any $cp \in C^A$: $cp = (P^{SA}_{cp}, T^{SA}_{cp}, M^{SA}_{cp}, F^{SA}_{cp}, \ell^{SA}_{cp})$, and let $N = \underline{cflat}(CA) = (P, T, C, F, \ell)$ and $i = \underline{source}(N)$. (CS, CA) is consistent if and only if*

(1) $(\forall t : t \in \underline{start}(N) \cup \underline{stop}(N) : \ell(t) \in \alpha(CS))$,

(2) *N is a sound C-net, i.e., $N \in \mathcal{C}$,*
(3) *$N \leq_{pj} CS$, and*
(4) *$(\forall\, cp : cp \in C^A : P_{cp}^{SA} \setminus \{\underline{source}(cp), \underline{sink}(cp)\}$ is activation safe in $(N, [i]))$, i.e., there is no multiple activation.*

Definition 34 gives the minimal set of requirements any component should satisfy. The first requirement states that the start and stop transitions of the flattened architecture have visible labels that appear in the component specification, i.e., it is not allowed to activate or deactivate a component by new methods. The flattened architecture, i.e., the functionality guaranteed by the architecture provided the correct operation of subcomponents, is sound. The flattened architecture is a subclass of the component specification with respect to projection inheritance. Finally, we require subcomponents to be started and stopped correctly, no multiple activation is allowed. Note that after terminating the subcomponent it may be activated again.

Both Figure 2 and Figure 3 show examples of components which are consistent.

The component shown in Figure 8 is not consistent for the following two reasons. First of all, the flattened architecture is not sound. Suppose that the method *switch_off* is initiated directly after inserting a coin. The subcomponent *brewing_facility* can be deactivated immediately. However, the *coin_handler* cannot be deactivated and will send a request to the *brewing_facility*, the *brewing_facility* will not respond to the request, and the machine will deadlock. Another reason for inconsistency is the fact that the *brewing_facility* sends an OK to the *coin_handler* before actually serving coffee. Therefore, one can insert a new coin before completely handling the previous request. This behavior does not invalidate the soundness requirement but yields a flattened architecture which is not a subclass of the original architecture.

The alternative component shown in Figure 11 does not have these deficiencies and is consistent. This component deactivates the *coin_handler* before deactivating the *brewing_facility*. Moreover, the coffee is served before the *coin_handler* is notified.

From the requirements stated in Definition 34, we can derive that the architecture of a component has a structure similar to a C-net, i.e., one unique source place and one unique sink place.

Lemma 35 *Let (CS, CA) be a consistent component with $CA = (P^A, T^A, C^A, F^A, \ell^A)$. There is precisely one $i \in P^A$ such that $\{t \in T^A \mid (t, i) \in F^A\} \cup \{(cp, l) \in C^A \times L_v \mid ((cp, l), i) \in F^A\} = \emptyset$ and precisely one $o \in P^A$ such that $\{t \in T^A \mid (o, t) \in F^A\} \cup \{(cp, l) \in C^A \times L_v \mid (o, (cp, l)) \in F^A\} = \emptyset$.*

PROOF. Since $\underline{cflat}(CA)$ is a C-net there is a place $i = \underline{source}(\underline{cflat}(CA))$. Clearly, $\{t \in T^A \mid (t, i) \in F^A\} \cup \{(cp, l) \in C^A \times L_v \mid ((cp, l), i) \in F^A\} = \emptyset$. For any other place, it is easy to show that $\underline{cflat}(CA)$ adds at least one input arc.

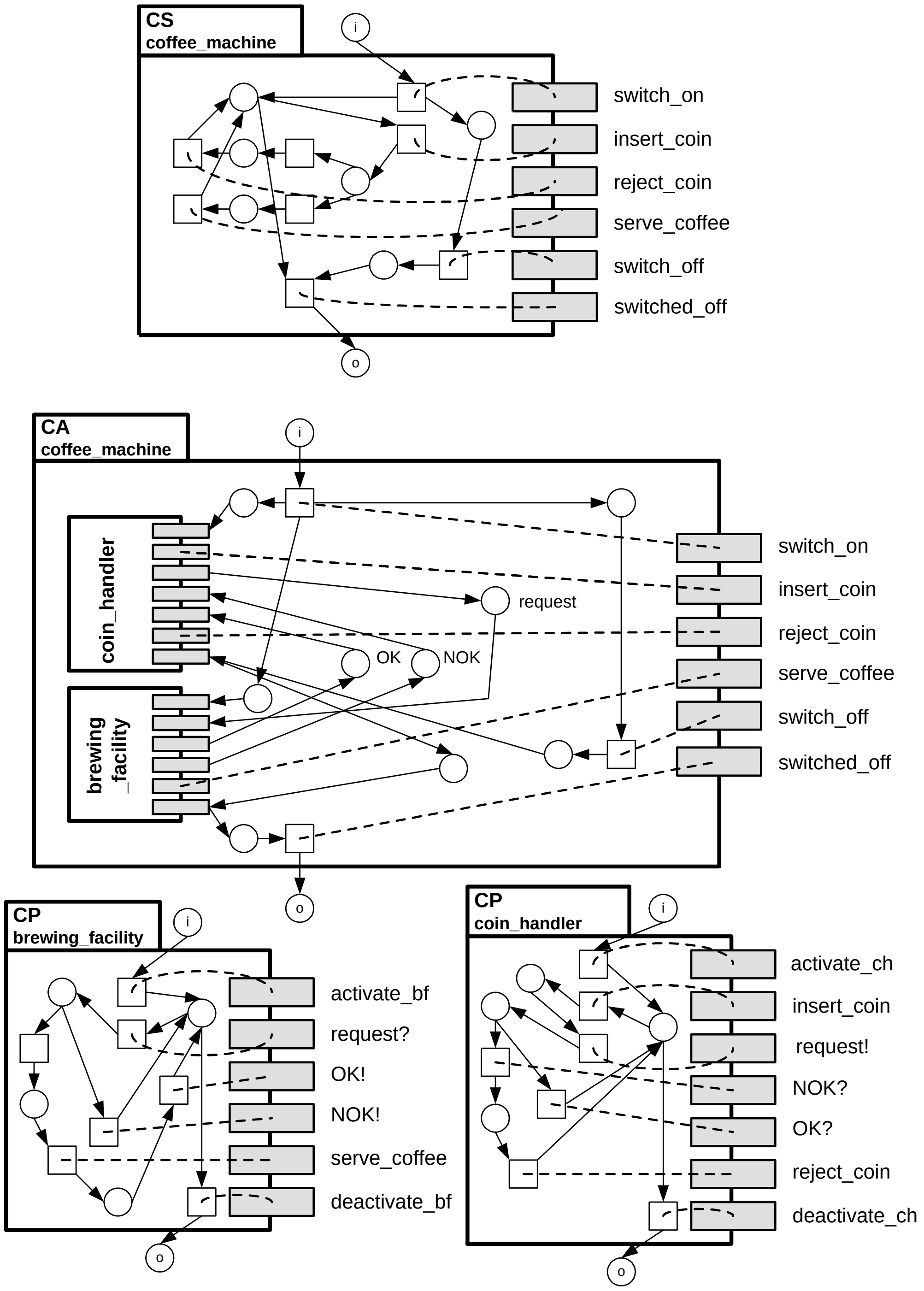


Fig. 11. A consistent version of the *coffee_machine* component: The two subcomponents are deactivated sequentially and coffee is served before the acknowledgement is sent.

Similarly, it can be shown that there is precisely one sink place. □

Since there is one source/sink place in the architecture of a component, we can define the functions $\underline{source}$, $\underline{sink}$, and $\underline{strip}$ in a straightforward manner for the architecture of a consistent component.

The first requirement in Definition 34 can be checked by simply inspecting the labels of start and stop transitions. The second requirement can be checked using the result stated in Theorem 17. The third requirement is either guaranteed by sticking to the inheritance-preserving transformation rules or by deploying a branching bisimulation checker (e.g., the inheritance checker in Wo*fl*an [35]). The last requirement (activation safeness) does not correspond to well-established notions such as liveness, safeness, and bisimulation and may be hard to check since there are no efficient analysis techniques/tools to verify this requirement. Therefore, we introduce a stronger requirement which can be validated syntactically (i.e., based on the structure of the flattened net). This requirement states that there is not a path from a transition inside one of the subcomponents to one of its start transitions not containing one of its stop transitions, i.e., the topology of the net guarantees that a subcomponent cannot trigger itself indirectly before it is deactivated. In other words: there is no *self triggering*. The following property defines the absence of self triggering and shows that the absence of self triggering assures that there is no multiple activation.

Proposition 36 (Self triggering) *Let be (CS, CA) be a component satisfying the first three requirements stated in Definition 34 (i.e., proper start/stop labels, N is a sound C-net, and N is a subclass of CS). If $(\forall\, cp, t, t' : cp \in C^A \wedge t \in T^{SA}_{cp} \wedge t' \in \underline{start}(cp)$: all non-trivial directed paths in N from t to t' contain at least one occurrence of a transition in $\underline{stop}(cp)$) and $(\forall\, cp : cp \in C^A : (\cap\, t : t \in \underline{start}(cp) : \overset{N}{\bullet} t) \neq \emptyset)$ (i.e., start transitions share input places), then (CS, CA) is consistent.*

PROOF. To prove this property, we need to show that each subcomponent (i.e., component placeholder) is activation safe. Let $cp \in C^A$ be an arbitrary component. Let $P_{cp} = P^{SA}_{cp} \backslash \{\underline{source}(cp), \underline{sink}(cp)\}$ be the set of internal places of this component. We need to prove that P_{cp} is activation safe in $(N, [i])$. We use proof by contradiction, i.e., we assume that there is a firing sequence σ such that $(N, [i])\,[\sigma\rangle\,(N, s)$, $t \in \underline{start}(cp)$, $(N, s)[t\rangle$, and $p \in P_{cp}$ is marked in s. Without loss of generality, we further assume that s was the first state in the sequence having these properties (i.e., a start transition is enabled while a place in P_{cp} is marked). Partition the sequence σ in two subsequences σ_1 and σ_2 such that σ_2 contains all firings since the last firing of a transition in $\underline{stop}(cp)$, i.e., σ_1 is either empty or ends with the last firing of a transition in $\underline{stop}(cp)$. The first sequence ends in state s' (i.e., $(N, [i])\ [\sigma_1\rangle\ (N, s')$). Note that in s' all places in P_{cp} are empty. (Otherwise there would have been a prefix of σ containing the anomaly.) Now we concentrate on the second subsequence: $(N, s')\ [\sigma_2\rangle\ (N, s)$. In this sequence no transition in $\underline{stop}(cp)$ fires. Therefore, we remove all transitions $\underline{stop}(cp)$ from N and name the new net N'. Note that $(N', s')[\sigma_2\rangle(N', s)$. The requirement that all non-trivial directed paths in N from a transition inside cp to one of the start transitions in cp contain at least one of the stop transitions in cp implies that we can partition the transitions of N' in two subsets T_X and T_Y such that $\{t \in T \backslash T_k \mid t \overset{N'}{\bullet} \cap \overset{N'}{\bullet} \underline{start}(cp) \neq \emptyset\} \subseteq T_X$, $T_k \subseteq T_Y$, and $\overset{N'}{\bullet} T_X \cap T_Y \overset{N'}{\bullet} = \emptyset$ because all stop transitions have been removed. Now we apply the well-known exchange lemma (see for example page 23 in [13]) which allows us to project σ_2 onto the transitions in T_X and T_Y: σ_{2X} and σ_{2Y}. Since $\overset{N'}{\bullet} T_X \cap T_Y \overset{N'}{\bullet} = \emptyset$, the exchange lemma shows that we can first execute σ_{2X} followed by σ_{2Y}. Let state s'' be the state after executing σ_{2X}, i.e., $(N', s') [\sigma_{2X}\rangle (N', s'')$. It is easy to see that in s'' at least one of the input places of the start transitions of cp contains multiple tokens, because start transitions share input places. (Note that σ_{2Y} marks a place in P_{cp}, i.e., fires at least one start transition of cp, and also enables a start transition of cp without adding any new tokens to the input places.) Therefore, the safeness property is violated. The sequence composed of σ_1 followed by σ_{2X} is also possible in $(N, [i])$. Therefore, $(N, [i])$ cannot be a sound C-net and we find a contradiction. $\square$

Property 36 shows that the only way that a subflow becomes activated multiple times (i.e., the place is not activation safe), is through self triggering. Note that in none of the components presented thus far there is any self-triggering. Therefore, each of the components shown in figures 2, 3, and 11 is activation safe.

A *system architecture* consists of a set of components where components are plugged into placeholders of other components. In Section 1 we introduced a system architecture composed of the *channel* (Figure 2) and *message_handler* (Figure 3) components.

Definition 37 (System architecture) *Let C be a set of components with for any $c \in C$, $c = (CS_c, CA_c)$, $CS_c = (P_c^S, T_c^S, M_c^S, F_c^S, \ell_c^S)$, $CA_c = (P_c^A, T_c^A, C_c^A, F_c^A, \ell_c^A)$, and $LC = \{(c, cp) \mid c \in C \wedge cp \in C_c^A\}$. A* system architecture $(C, \underline{cmap})$ *is a set of components C and a mapping $\underline{cmap} : LC \to C$.*

A component can not be plugged into more than one placeholder, i.e., it is not possible to have two separate components sharing a third component. In addition, recursive structures are not allowed. Moreover, there should be one *top-level* component which contains all other components. The latter requirement has been added for presentation purposes and does not limit the application of the framework: Any set of components can be embedded into one component. A system architecture satisfying these requirements is called *well-formed*.

Definition 38 (Well-formed) *Let $(C, \underline{cmap})$ be a system architecture such that for any $c \in C$: $c = (CS_c, CA_c)$, $CS_c = (P_c^S, T_c^S, M_c^S, F_c^S, \ell_c^S)$, and $CA_c = (P_c^A, T_c^A, C_c^A, F_c^A, \ell_c^A)$. C is well-formed if and only if the relation $R = \{(c, c') \in C \times C \mid (c, cp) \in LC \wedge \underline{cmap}(c, cp) = c'\}$ describes a rooted directed acyclic* *graph.*[5]

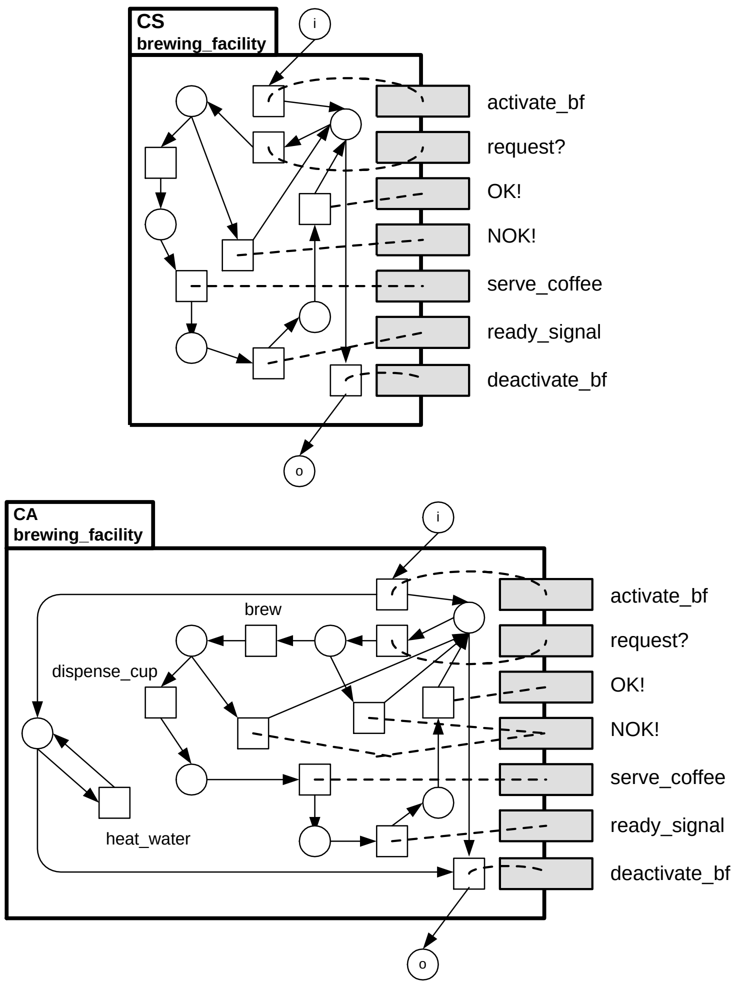
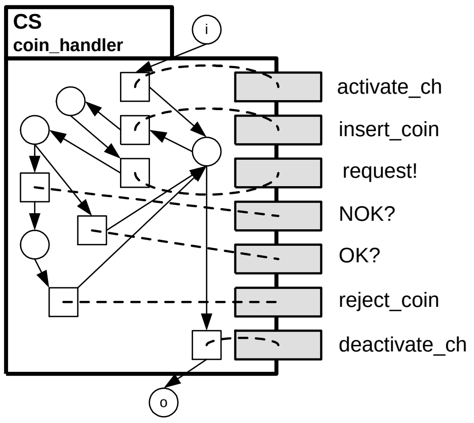


Fig. 12. The component *brewing_facility*.

Clearly the system architecture introduced in Section 1 is well-formed: The only placeholder in *channel* is mapped onto the component *message_handler*. Let us also consider the system architecture for a coffee machine. The component shown in Figure 11 is the top-level component. The architecture of the top-level component has two component placeholders. The placeholder *brewing_facility* is mapped onto the component *brewing_facility* shown in Figure 12 and the placeholder *coin_handler* is mapped onto a component with a component specification and architecture identical to the C-net describing the placeholder (see Figure 13). Note that both sub-components are closed, i.e., the system architecture for a coffee machine has two levels and comprises three components. Clearly, this simple system architecture is well-formed.

[5] A directed acyclic graph is rooted if there is a node r such that every node of the graph can be reached by a directed path from r.



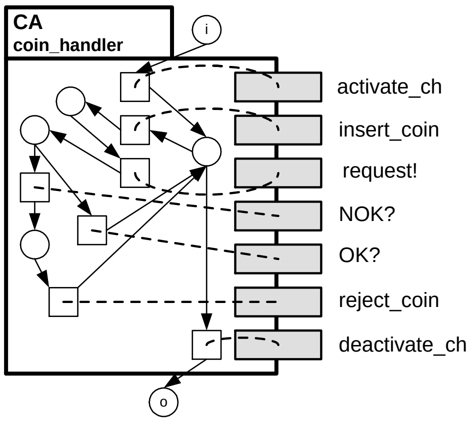


Fig. 13. The component *coin_handler*.

Similar to consistency at a component level, we can define consistency at the level of a system architecture.

Definition 39 (Consistent) *Let* $(C, \underline{cmap})$ *be a well-formed system architecture such that for any* $c \in C$*:* $c = (CS_c, CA_c)$*,* $CS_c = (P^S_c, T^S_c, M^S_c, F^S_c, \ell^S_c)$*, and* $CA_c = (P^A_c, T^A_c, C^A_c, F^A_c, \ell^A_c)$*.* $(C, \underline{cmap})$ *is* consistent *if and only if*

(1) *each component* $c \in C$ *is consistent, and*
(2) *for all* $c \in C$*,* $c' \in C$*, and* $cp \in C^A_c$ *such that* $\underline{cmap}(c, cp) = c'$*:*
  (a) $CS_{c'} \leq_{pj} cp$*, and*
  (b) $(\forall\, t : t \in \underline{start}(CS_{c'}) \cup \underline{stop}(CS_{C'}) : \ell^S_{c'}(t) \in \alpha(cp))$.

A well-formed system architecture is consistent if the individual components are consistent and appropriate components are plugged into the placeholders, i.e., if a component is plugged into the placeholder, then its specification should be a subclass of the C-net specifying the placeholder and the plugged-in component should not introduce other methods for activating and deactivating components. The latter requirement has been added to avoid the activation/deactivation of a component by methods not present in the C-net specifying the placeholder, i.e., without this requirement the subcomponents could easily deadlock or lead to unbounded behavior.

Consider the system architecture for the coffee machine composed of the top-level component shown in Figure 11, the component *brewing_facility* shown in Figure 12, and the component *coin_handler* shown in Figure 13. Each of the three components is consistent. Note that the component *brewing_facility* offers the method *ready_signal* to its environment, i.e., the component generates a signal every time a cup of coffee has been served and thus offers more functionality than needed. Also note that the architecture of the component *brewing_facility* shows details not present in the component specification, e.g., the internal steps *brew*, *dispense_cup*, and *heat_water*. The steps *brew* and *dispense_cup* are executed after the request for a coffee is received. In-between these steps the brewing facility can produce an error which is reported via method *NOK!*. The internal step *heat_water* is executed periodically (e.g., driven by a thermostat) and in parallel with the handling of requests. The component specification of *brewing_facility* is a subclass of the component placeholder in Figure 11. The component specification of *coin_handler* coincides with the corresponding placeholder and, consequently, is also a subclass. Therefore, the system architecture for the coffee machine is consistent.

Clearly the system architecture introduced in Section 1 is also consistent.

A consistent system architecture satisfies a number of requirements. In the remainder of this paper, we will concentrate on the question whether these requirements imply the correct operation of the entire system, i.e., *Is it guaranteed that the system actually realizes the functionality suggested by the specification of the top-level component?*

4 Compositionality results

Based on the framework introduced in the previous section, we focus on the question whether consistency guarantees the correct operation of the whole system architecture. To be more precise, we will show that:

- the flattened *system* architecture is a sound C-net, i.e., if all component placeholders are replaced by actual components, then the resulting system is free of deadlocks and other anomalies,
- the flattened system architecture is a subclass of the specification of the top-level component under projection inheritance, i.e., the system realizes the desired behavior.

To prove these statements, we first formulate and prove a rather general theorem. This theorem addresses the notion of compositionality in the context of projection inheritance.

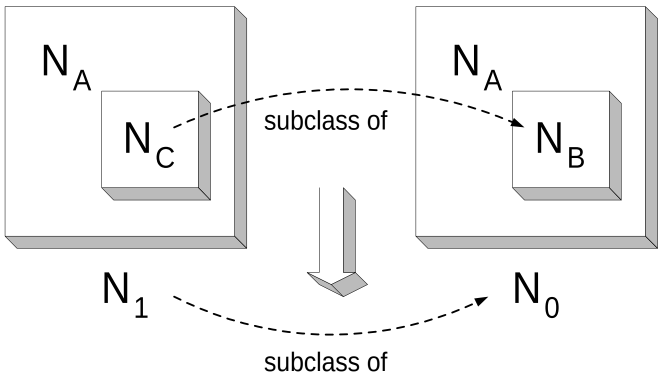


Fig. 14. The essence of Theorem 40: if N_C^W is a subclass of N_B^W, then N_1 is a subclass of N_0.

Figure 14 illustrates the essence of Theorem 40: Consider a sound C-net N_0 composed of N_A and N_B. N_A and N_B communicate through a set of common places $P_A \cap P_B$. N_B is chosen in such a way that if we remove the places $P_A \cap P_B$ and add a source and a sink place, we obtain a sound C-net N_B^W. In addition, it is assumed that there is no multiple activation. Moreover, there are three additional P/T nets N_C, N_C^W, and N_1. N_1 is composed of N_A and N_C. The connections between N_A and N_C in N_1 are essentially the same as the connections between N_A and N_B in N_0, e.g., $P_A \cap P_C = P_A \cap P_B$ (see Theorem 40 for details). Moreover, N_C is chosen in such a way that if we remove the places $P_A \cap P_C$ and add a source and a sink place, we obtain a sound C-net N_C^W *which is a subclass of* N_B^W *under projection inheritance*. Under these conditions N_1 is guaranteed to be sound and a subclass of N_0. In other words: Theorem 40 shows that inheritance is some kind of congruence under the composition of C-nets.

Theorem 40 (Compositionality of projection inheritance) *Let* $N_0 = (P_0, T_0, M_0, F_0, \ell_0)$, $N_1 = (P_1, T_1, M_1, F_1, \ell_1)$, $N_A = (P_A, T_A, M_A, F_A, \ell_A)$, $N_B = (P_B, T_B, M_B, F_B, \ell_B)$, $N_C = (P_C, T_C, M_C, F_C, \ell_C)$, $N_B^W = (P_B^W, T_B^W, M_B^W, F_B^W, \ell_B^W)$, *and* $N_C^W = (P_C^W, T_C^W, M_C^W, F_C^W, \ell_C^W)$ *be labeled P/T-nets. If*

(1) N_0 *is a sound C-net in* $\mathcal{C}$ *with source place* $i = \underline{source}(N_0)$ *and sink place* $o = \underline{sink}(N_0)$,
(2) $N_0 = N_A \cup N_B$ *is well defined,*
(3) $N_1 = N_A \cup N_C$ *is well defined,*
(4) $T_A \cap T_B = \emptyset$,
(5) $T_A \cap T_C = \emptyset$,
(6) $P_A \cap P_B = P_A \cap P_C$,
(7) N_B^W *is a sound C-net in* $\mathcal{C}$ *such that* $\underline{strip}(N_B^W) = (P_B \backslash P_A, T_B, M_B, F_B \cap ((P_B^W \times T_B^W) \cup (T_B^W \times P_B^W)), \ell_B)$, $i_B = \underline{source}(N_B^W)$, $o_B = \underline{sink}(N_B^W)$, *and* $\{i_B, o_B\} \cap P_0 = \emptyset$,
(8) N_C^W *is a sound C-net in* $\mathcal{C}$ *such that* $\underline{strip}(N_C^W) = (P_C \backslash P_A, T_C, M_C, F_C \cap$

$((P_C^W \times T_C^W) \cup (T_C^W \times P_C^W)), \ell_C)$, $i_C = \underline{source}(N_C^W)$, $o_C = \underline{sink}(N_C^W)$, and $\{i_C, o_C\} \cap P_1 = \emptyset$,

(9) $(\forall\, t : t \in \underline{start}(N_C^W) \cup \underline{stop}(N_C^W) : \ell_C^W(t) \in \alpha(N_B^W))$, i.e., no new labels are introduced for the start and stop transitions,

(10) $(\forall\, t : t \in T_B \wedge \ell_B(t) = \tau : (\overset{N_0}{\bullet} t \cap P_A = \emptyset) \wedge (t \overset{N_0}{\bullet} \cap P_A = \emptyset))$, i.e., transitions with a τ label are not connected to "outside places",

(11) $(\forall\, t : t \in T_C \wedge \ell_1(t) \notin \alpha(N_B^W) : (\overset{N_1}{\bullet} t \cap P_A = \emptyset) \wedge (t \overset{N_1}{\bullet} \cap P_A = \emptyset))$, i.e., transitions with a "new label" are not connected to outside places,

(12) $(\forall\, t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : (\overset{N_0}{\bullet} t \cap P_A = \overset{N_1}{\bullet} t' \cap P_A) \wedge (t \overset{N_0}{\bullet} \cap P_A = t' \overset{N_1}{\bullet} \cap P_A))$,

(13) P_B^W is activation safe in $(N_0, [i])$, and

(14) $N_C^W \leq_{pj} N_B^W$,

then N_1 is a sound C-net in $\mathcal{C}$ such that $N_1 \leq_{pj} N_0$.

PROOF. The proof consists of three parts. First, we provide some useful observations. Then, we show that there is a branching bisimulation between $(N_0, [i])$ and $\tau_I(N_1, [i])$ ($I = \alpha(N_1) \backslash \alpha(N_0)$). Finally, we show that N_1 is a sound C-net and conclude that $N_1 \leq_{pj} N_0$ using the branching bisimulation.

Part A

The following observations are useful for the remainder of the proof:

(1) Since $N_C^W \leq_{pj} N_B^W$, $\alpha(N_B^W) \subseteq \alpha(N_C^W)$ and there is a branching bisimulation $\mathcal{R}_{BC}$ such that $(N_B^W, [i_B])\ \mathcal{R}_{BC} \tau_I(N_C^W, [i_C])$ with $I = \alpha(N_C^W) \backslash \alpha(N_B^W) = \alpha(N_1) \backslash \alpha(N_0)$. Without loss of generality we assume that $\mathcal{R}_{BC} \subseteq \{((N_B^W, s_B), (N_C^W, s_C)) \mid s_B \in [N_B^W, [i_B]\rangle \wedge s_C \in [N_C^W, [i_C]\rangle\}$.
$\diamond$ This follows directly from the definition of projection inheritance.

(2) $\{\ell_B(t) \mid t \in \underline{start}(N_B^W)\} = \{\ell_C(t) \mid t \in \underline{start}(N_C^W)\}$, i.e., the sets of start labels coincide.
$\diamond$ Since $(\forall\, t : t \in \underline{start}(N_C^W) \cup \underline{stop}(N_C^W) : \ell_C^W(t) \in \alpha(N_B^W))$ start transitions in both N_B^W and $\tau_I(N_C^W)$ are visible (i.e., not τ). Since all actions enabled in $[i_B]$ respectively $[i_C]$ are visible and $(N_B^W, [i_B])\ \mathcal{R}_{BC} \tau_I(N_C^W, [i_C])$, these action sets have to match and therefore the sets of start labels coincide.

(3) $\{\ell_B(t) \mid t \in \underline{stop}(N_B^W)\} = \{\ell_C(t) \mid t \in \underline{stop}(N_C^W)\}$, i.e., the sets of stop labels coincide.
$\diamond$ For similar reasons the stop transitions are visible. If there is a stop transition in one net with a label not appearing in the other net as a label of a stop transition, then it is easy to show that this is in contradiction with $(N_B^W, [o_B])$ $\mathcal{R}_{BC}\ \tau_I(N_C^W, [o_C])$.

(4) $(\forall\, t, t' : t \in T_B \wedge t' \in T_B \wedge \ell_B(t) = \ell_B(t') : (\overset{N_0}{\bullet} t \cap P_A = \overset{N_0}{\bullet} t' \cap P_A) \wedge (t \overset{N_0}{\bullet} \cap P_A = t' \overset{N_0}{\bullet} \cap P_A))$, i.e., transitions in T_B with identical labels have identical

effects on the interface $P_A \cap P_B$.
$\diamond$ If both transitions have a τ label, then there are no connections to the interface $P_A \cap P_B$. If the transitions have a visible label, then there is a corresponding transition in N_C. Since the connections of this transition in N_C to places in $P_A \cap P_B$ are identical to those of t and t', the external connections of t and t' have to match.

(5) $(\forall\, t, t' : t \in T_C \wedge t' \in T_C \wedge \ell_C(t) = \ell_C(t') : (\overset{N_1}{\bullet} t \cap P_A = \overset{N_1}{\bullet} t' \cap P_A) \wedge (t \overset{N_1}{\bullet} \cap P_A = t' \overset{N_1}{\bullet} \cap P_A))$, i.e., transitions in T_C with identical labels have identical effects on the interface $P_A \cap P_C$.
$\diamond$ If both transitions have a τ label or a label not used in N_B, then there are no connections to the interface $P_A \cap P_B$. If the transitions have a visible label used in N_B, then there is a corresponding transition in N_B. Since the connections of this transition in N_B to places in $P_A \cap P_C$ are identical to those of t and t', the external connections of t and t' have to match.

(6) For any $t \in \underline{start}(N_B^W)$, there exists a $t' \in \underline{start}(N_C^W)$ such that $\ell_B(t) = \ell_C(t')$ and $\overset{N_0}{\bullet} t = \overset{N_1}{\bullet} t'$, and for any $t \in \underline{stop}(N_C^W)$, there exists a $t' \in \underline{stop}(N_B^W)$ such that $\ell_C(t) = \ell_B(t')$ and $t \overset{N_1}{\bullet} = t' \overset{N_0}{\bullet}$.
$\diamond$ This follows directly from the requirement that all start and stop labels are visible, the sets of start labels coincide, the sets of stop labels coincide, and transitions in different nets with identical commonly visible labels have identical sets of input/output places.

(7) N_0, N_1, N_B^W, and N_C^W completely determine N_A, N_B, and N_C.
$\diamond$ $N_A = N_0 \cap N_1$, $N_B = (\overset{N_0}{\bullet} T_B^W \cup T_B^W \overset{N_0}{\bullet}, T_B^W, M_B^W, F_0 \cap ((P_B \times T_B) \cup (T_B \times P_B)), \ell_B^W)$, and $N_C = (\overset{N_1}{\bullet} T_C^W \cup T_C^W \overset{N_1}{\bullet}, T_C^W, M_C^W, F_1 \cap ((P_C \times T_C) \cup (T_C \times P_C)), \ell_C^W)$.

(8) Any marking $s_0 \in [N_0, [i]\rangle$ can be partitioned into s_A and s_B such that $s_0 = s_A + s_B$, $s_A \in \mathcal{B}(P_A)$, $s_B \in \mathcal{B}(P_0 \backslash P_A)$, and $s_B = \mathbf{0}$ or $s_B \in [N_B^W, [i_B]\rangle$.
$\diamond$ Initially, s_B is empty. (Note that $i \in P_A$.) The only way to mark places in $P_0 \backslash P_A$ is to fire a transition in $\underline{start}(N_B^W)$. However, P_B^W is activation safe. Therefore, the start transitions are blocked until $P_B \backslash P_A = P_0 \backslash P_A$ is empty again and it is not possible to reach states outside $[N_B^W, [i_B]\rangle$.

Part B
Based on $\mathcal{R}_{BC}$ and N_0, N_1, N_B^W, and N_C^W as defined above. We define $\mathcal{R}_{01}$ as follows: $\mathcal{R}_{01} = \{((N_0, s_A + s_B), \tau_I(N_1, s_A + s_C)) \mid s_A \in \mathcal{B}(P_A) \wedge s_B \in \mathcal{B}(P_0 \backslash P_A) \wedge s_C \in \mathcal{B}(P_1 \backslash P_A) \wedge s_A + s_B \in [N_0, [i]\rangle \wedge s_A + s_C \in [N_1, [i]\rangle \wedge ((s_B = \mathbf{0} \wedge s_C = \mathbf{0}) \vee ((N_B^W, s_B) \mathcal{R}_{BC} \tau_I(N_C^W, s_C)))\}$. We show that $\mathcal{R}_{01}$ is a branching bisimulation and that $(N_0, [i]) \mathcal{R}_{01} \tau_I(N_1, [i])$.

Consider two markings $s_0 \in [N_0, [i]\rangle$ and $s_1 \in [N_1, [i]\rangle$ such that $(N_0, s_0)\mathcal{R}_{01}$ $\tau_I(N_1, s_1)$. The bags s_0 and s_1 can be partitioned as in the definition of $\mathcal{R}_{01}$, i.e., $s_0 = s_A + s_B$, $s_1 = s_A + s_C$, $s_A \in \mathcal{B}(P_A)$, $s_B \in \mathcal{B}(P_0 \backslash P_A)$, $s_C \in \mathcal{B}(P_1 \backslash P_A)$. For these two markings we will verify the three requirements stated in the definition of branching bisimilarity.

(1) Assume that $t \in T_0$ is such that $(N_0, s_0)\ [\ell_0(t)\rangle\ (N_0, s_0')$. Bag s_0' can be partitioned into s_A' and s_B' as before. We need to prove that there exist s_1', s_1'' such that $(N_1, s_1) \Longrightarrow (N_1, s_1'')\ [(\ell_0(t))\rangle\ (N_1, s_1') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}\ (N_1, s_1')$.

- If $t \in T_A$, then t is also enabled in (N_1, s_1) and firing t only affects places in P_A because $\overset{N_0}{\bullet} t \cup t \overset{N_0}{\bullet} = \overset{N_1}{\bullet} t \cup t \overset{N_1}{\bullet} \subseteq P_A$. Moreover, $\ell_0(t) = \ell_1(t)$. Therefore, $s_1'' = s_1$ and $s_1' = s_A' + s_C$ are such that $(N_1, s_1) \Longrightarrow (N_1, s_1'')\ [(\ell_0(t))\rangle\ (N_1, s') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}(N_1, s_1')$.
- If $t \notin T_A$, then $t \in T_B$.
  - Assume $t \in \underline{start}(N_B^W)$. Since P_B^W is activation safe in $(N_0, [i])$, $s_B = \mathbf{0}$. Moreover, $s_C = \mathbf{0}$, because $s_B = \mathbf{0}$, $(N_0, s_0)\mathcal{R}_{01}\tau_I(N_1, s_1)$, and there is no s_C such that $(N_B^W, \mathbf{0})\mathcal{R}_{BC}(\tau_I(N_C^W, s_C))$. Since transition $t \in \underline{start}(N_B^W)$, each place in $\overset{N_0}{\bullet} t$ is marked in both s_0 and s_1. Moreover, $\ell_0(t) \neq \tau$. Clearly, there is a $t' \in T_C$ such that $\ell_0(t) = \ell_1(t')$ and $\overset{N_1}{\bullet} t' = \overset{N_0}{\bullet} t \subseteq \mathcal{B}(P_A)$. Since s_0 and s_1 are identical with respect to the places in P_A, t' is also enabled in (N_1, s_1). Moreover, the result of firing t' is identical to t with respect to the places in P_A. Let s_C' be such that $(N_C^W, [i_C])\,[\ell_C(t')\rangle\,(N_C^W, s_C')$ and $(N_B^W, s_B')\mathcal{R}_{BC}\tau_I(N_C^W, s_C')$. Such a s_C' exists because $(N_B^W,\ [i_B])\ \mathcal{R}_{BC}\ \tau_I(N_C^W, [i_C])$. It is easy to see that $s_1'' = s_1$ and $s_1' = s_A' + s_C'$ are such that $(N_1, s_1) \Longrightarrow (N_1, s_1'')[(\ell_0(t))\rangle(N_1, s_1') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}(N_1, s_1')$.
  - Assume $t \in \underline{stop}(N_B^W)$. Clearly, $s_B \neq \mathbf{0}$. Hence, $(N_B^W, s_B)\ \mathcal{R}_{BC}\ (\tau_I(N_C^W,\ s_C))$. Transition t is also enabled in (N_B^W, s_B). Since N_B^W is sound, $s_B \in [N_B^W, [i_B]\rangle$, and $t \in \underline{stop}(N_B^W)$, we deduce that $(N_B^W, s_B)\ [\ell(t)\rangle\ (N_B^W, [o_B])$. Hence, $s_B' = \mathbf{0}$. Since $(N_B^W, s_B)\mathcal{R}_{BC}(\tau_I(N_C^W, s_C))$, it is straightforward to show that in (N_C^W, s_C) a sequence consisting of zero or more silent steps can be executed followed by the firing of a transition t' such that $\ell_0(t) = \ell_1(t')$. Let s_C' be the resulting marking. Since N_C^W is sound, $s_C \in [N_C^W, [i_C]\rangle$, and $t' \in \underline{stop}(N_C^W)$, $s_C' = [o_C]$. Clearly, the same sequence can be executed in (N_1, s_1) leading to s_1'. Note that in s_1' only places in P_A are marked. Since the effects of transitions t in N_0 and t' in N_1 on the places in P_A are identical, $s_1' = s_0'$. Therefore, there are s_1'' and s_1' such that $(N_1, s_1) \Longrightarrow (N_1, s_1'')\ [(\ell_0(t))\rangle\ (N_1, s') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}(N_1, s_1')$.
  - Assume $t \in T_B \backslash (\underline{start}(N_B^W) \cup \underline{stop}(N_B^W))$. Since $s_B \neq \mathbf{0}$, $(N_B^W, s_B)\ \mathcal{R}_{BC}\ (\tau_I(N_C^W, s_C))$.

    If $\ell_0(t) = \tau$, then choose $s_1' = s_1'' = s_1$. It is easy to see that $(N_1, s_1) \Longrightarrow (N_1, s_1'')\ [(\ell_0(t))\rangle\ (N_1, s') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}(N_1, s_1')$.

    If $\ell_0(t) \neq \tau$, then it is straightforward to show that in (N_C^W, s_C) a sequence consisting of zero or more silent steps can be executed followed by the firing of a transition t' such that $\ell_0(t) = \ell_1(t')$.

Let s'_C be the resulting marking. Clearly, $(N^W_B, s'_B)\mathcal{R}_{BC}(\tau_I(N^W_C, s'_C))$ and $s'_C \in \mathcal{B}(P_1 \setminus P_A)$, i.e., s'_C does not mark o_C. The same sequence can be executed in (N_1, s_1) leading to s'_1. The effect of the execution of t' on the places in P_A is identical to the effect of t on the places in P_A , i.e., $(\forall\, t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : (\overset{N_0}{\bullet} t \cap P_A = \overset{N_1}{\bullet} t' \cap P_A) \wedge (t \overset{N_0}{\bullet} \cap P_A = t' \overset{N_1}{\bullet} \cap P_A))$. Therefore, there are s''_1 and s'_1 such that $(N_1, s_1) \Longrightarrow (N_1, s''_1)\ [(\ell_0(t))\rangle\ (N_1, s') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s''_1) \wedge (N_0, s'_0)\mathcal{R}_{01}(N_1, s'_1)$.

(2) Assume that $t \in T_1$ is such that $(N_1, s_1)\ [\ell_1(t)\rangle\ (N_1, s'_1)$. We need to prove that there exist s'_0, s''_0 such that $(N_0, s_0) \Longrightarrow (N_0, s''_0)\ [(\ell_1(t))\rangle\ (N_0, s'_0) \wedge (N_0, s''_0)\mathcal{R}_{01}\ (N_1, s_1) \wedge (N_0, s'_0)\mathcal{R}_{01}(N_1, s'_1)$. The proof is almost identical to the proof in the other direction. The only issue which should be noted is that if $t \in \underline{start}(N^W_C)$ is enabled in (N_1, s_1), then $s_C = \mathbf{0}$: Because t is also enabled in (N_0, s_0) and P^W_B is activation safe, $s_B = \mathbf{0}$. Moreover, there is no s_C such that $(N^W_B, \mathbf{0})\ \mathcal{R}_{BC}\ (\tau_I(N^W_C, s_C))$. Hence, $s_C = \mathbf{0}$.

(3) Assume $\downarrow s_0$. We need to prove that $\Downarrow s_1$. $\downarrow s_0$ implies that $s_0 = [o]$, $s_A = [o]$, and $s_B = \mathbf{0}$. If $s_C = \mathbf{0}$, then $s_1 = [o]$ and $\Downarrow s_1$ (in fact $\downarrow s_1$). It is not possible that $s_C \neq \mathbf{0}$, because this would imply that $(N^W_B, \mathbf{0})\mathcal{R}_{BC}\tau_I(N^W_C, s_C)$ which is not possible. Similarly, it can be shown that $\downarrow s_1$ implies $\Downarrow s_0$.

From the definition of $\mathcal{R}_{01}$ it follows that $(N_0, [i])\mathcal{R}_{01}\tau_I(N_1, [i])$.

Part C

It remains to be proven that N_1 is a sound C-net. It is easy to see that N_1 is a C-net: There is one source place i, one sink place o, and every node is on a path from i to o. To prove that N_1 is sound, consider an arbitrary marking $s_1 \in [N_1, [i]\rangle$. For this marking there is a counterpart s_0 in the original net (N_0) such that $s_0 \in [N_0, [i]\rangle$ and $(N_0, s_0)\mathcal{R}_{01}\tau_I(N_1, s_1)$. Using s_0 we verify the four requirements for soundness:

- $(N_1, [i])$ is safe because, for any place $p \in P_A$, $s_1(p) = s_0(p) \leq 1$, and there is a marking $s_C \in [N^W_C, [i_C]\rangle$ such that for any place $p \in P_1 \setminus P_A$: $s_1(p) = s_C(p) \leq 1$.
- Suppose that $o \in s_1$. Since N_0 is sound, $s_0 = [o]$. Since $(N_0, s_0)\mathcal{R}_{01}\ \tau_I(N_1, s_1)$, the other places in P_A are empty. The places in $P_1 \setminus P_A$ are also empty, because otherwise there would be a nonempty bag s_C such that $s_C \neq [o_B]$ and $(N^W_B, \mathbf{0})\mathcal{R}_{BC}\ \tau_I(N^W_C, s_C)$. Clearly this is not possible because from s_C it would be possible to fire a non-τ-labeled transition.
- From s_0 it is possible to reach the marking $[o]$ in N_0 because N_0 is sound. Since $(N_0, s_0) \sim_b \tau_I(N_1, s_1)$ it is possible to do the same in N_1 starting from s_1.
- To prove that there are no dead transitions in $(N_1, [i])$, we first consider transitions in T_A. Suppose a transition $t \in T_A$ is enabled in (N_0, s_0), then t is also enabled in (N_1, s_1). Since there are no dead transitions in $(N_0, [i])$, it is possible to enable any transition $t \in T_A$ starting from $(N_1, [i])$. Transitions in $T_1 \setminus T_A$ are not dead, because there are no dead transitions in $(N^W_C, [i_C])$.

Since N_1 is a sound C-net and $\mathcal{R}_{01}$ is a branching bisimulation, we conclude that $N_1 \leq_{pj} N_0$. $\square$

To show that a consistent well-formed system architecture actually provides the functionality assured by the specification of the top-level component, we define a function $\underline{aflat}$ to translate a system architecture into a labeled P/T net.

Definition 41 (Flattened architecture) *Let $(C, \underline{cmap})$ be a well-formed system architecture such that for any $c \in C$: $c = (CS_c, CA_c)$, $CS_c = (P_c^S, T_c^S, M_c^S, F_c^S, \ell_c^S)$, and $CA_c = (P_c^A, T_c^A, C_c^A, F_c^A, \ell_c^A)$. The corresponding flattened architecture is the labeled P/T net $\underline{aflat}(C, \underline{cmap})$ obtained by applying the following algorithm:*

Step 1 *c^t is the top level component, i.e., the root of the directed acyclic graph R mentioned in Definition 38.*
$CA = (P^A, T^A, C^A, F^A, \ell^A) := CA_{c^t}$
$\underline{hmap}(cp) := \underline{cmap}(c^t, cp)$ *for all* $cp \in C_{c^t}^A$

Step 2 *If $C^A = \emptyset$, then stop and output $\underline{aflat}(C, \underline{cmap}) = (P^A, T^A, rng(\ell^A), F^A, \ell^A)$, otherwise goto Step 3.*

Step 3 *Select a $cp \in C^A$.*
$c := \underline{hmap}(cp)$
$CA' = (P^{A'}, T^{A'}, C^{A'}, F^{A'}, \ell^{A'}) := \underline{strip}(CA_c)$
$P^{A''} := P^A \cup P^{A'}$
$T^{A''} := T^A \cup T^{A'}$
$C^{A''} := (C^A \backslash \{cp\}) \cup C_c^A$
$F^{A''} := (F^A \backslash (((\{cp\} \times L_v) \times P^A) \cup (P^A \times (\{cp\} \times L_v)))) \cup F^{A'} \cup \{(p, x) \in P^A \times dom(\ell^{A'}) \mid (p, (cp, \ell^{A'}(x))) \in F^A\} \cup \{(x, p) \in dom(\ell^{A'}) \times P^A \mid ((cp, \ell^{A'}(x)), p) \in F^A\}$
$dom(\ell^{A''}) := (dom(\ell^A) \backslash (\{cp\} \times L)) \cup dom(\ell^{A'})$.
For any $x \in dom(\ell^{A''})$: if $x \in dom(\ell^{A'})$, then $\ell^{A''}(x) := \ell^A(cp, \ell^{A'}(x))$, otherwise $\ell^{A''}(x) := \ell^A(x)$.
$CA'' := (P^{A''}, T^{A''}, C^{A''}, F^{A''}, \ell^{A''})$
$\underline{hmap}(cp') := \underline{cmap}(c, cp')$ *for all* $cp' \in C_c^A$
$CA := CA''$
Goto Step 2.

To flatten the system architecture, the placeholders in the top-level component are replaced by the architectures of the corresponding components. Then the newly introduced placeholders are replaced by the component architectures, etc., until there are only closed components.

Figure 4 shows the flattened system architecture introduced in Section 1, i.e., the system composed of the components shown in figures 2 and 3.

Note that the flattened architecture corresponds to the actual behavior of the system and that there are similarities with flattening other types of hierarchical Petri nets [19]. The following theorem uses the compositionality result of Theorem 40 to show that consistency implies the proper operation of the whole system.

Theorem 42 (Consistency implies soundness and conformance) *Let $(C, \underline{cmap})$ be a consistent well-formed system architecture with top-level component $c^t = (CS, CA)$. $\underline{aflat}(C, \underline{cmap})$ is a sound C-net and $\underline{aflat}(C, \underline{cmap}) \leq_{pj} CS$.*

PROOF. The algorithm specified in Definition 41 unfolds a component architecture $CA = (P^A, T^A, C^A, F^A, \ell^A)$ in a number of steps. We will show that at any point in time $\underline{cflat}(CA) \leq_{pj} CS_{c^t}$ using induction.

Initially, $CA = CA_{ct}$. Since the top-level component c^t is consistent, $\underline{cflat}(CA)$ is a sound C-net and $\underline{cflat}(CA) \leq_{pj} CS_{c^t}$ (see Definition 34).

Assume that $\underline{cflat}(CA) \in \mathcal{C}$, $\underline{cflat}(CA) \leq_{pj} CS_{c^t}$, and $cp \in C^A$, $c = \underline{hmap}(cp)$, $CA' = \underline{strip}(CA_c)$, and CA'' as defined in Step 3 of the algorithm. We will prove that CA'' is a sound C-net in $\mathcal{C}$ and $\underline{cflat}(CA'') \leq_{pj} \underline{cflat}(CA)$ using Theorem 40. Let $N_0 = \underline{cflat}(CA)$, $N_1 = \underline{cflat}(CA'')$, $N_B^W = cp$, and $N_C^W = \underline{cflat}(CA_c)$. It is easy to verify that N_0, N_1, N_A^W, and N_B^W satisfy the requirements stated in Theorem 40:

(1) N_0 is a sound C-net because we assume $\underline{cflat}(CA) \in \mathcal{C}$.
(2) $N_0 = N_A \cup N_B$ is well defined because the subnets do not share transitions.
(3) $N_1 = N_A \cup N_C$ is for the same reason well defined.
(4) $T_A \cap T_B = \emptyset$, see assumption on name clashes.
(5) $T_A \cap T_C = \emptyset$, see assumption on name clashes.
(6) $P_A \cap P_B = P_A \cap P_C$, follows directly from the construction.
(7) N_B^W is a sound C-net because placeholder $cp \in \mathcal{C}$.
(8) N_C^W is a sound C-net because c is consistent and therefore $\underline{cflat}(CA_c)$ is sound.
(9) Since $(C, \underline{cmap})$ is consistent, the set of labels used by $\underline{start}(cp)$ equals the set of labels used by $\underline{start}(CS_c)$. Moreover, since c is consistent, the set of labels used by $\underline{start}(\underline{cflat}(CA_c))$ also equals the set of labels used by $\underline{start}(CS_c)$. Since cp and $\underline{cflat}(CA_c)$ use the same set of labels for start transitions, the construction in Step 3 (which is purely based on labels) guarantees that the set of labels used by $\underline{start}(N_C^W)$ is a subset of $\alpha(N_B^W)$. Similar remarks hold for the labels of stop transitions. Hence, $(\forall t : t \in \underline{start}(N_C^W) \cup \underline{stop}(N_C^W) : \ell_C^W(t) \in \alpha(N_B^W))$.
(10) Since only transitions with non-τ labels in cp are connected to places in CA by the $\underline{cflat}$ function, $(\forall t : t \in T_B \wedge \ell_B(t) = \tau : (\overset{N_0}{\bullet} t \cap P_A = \emptyset) \wedge (t \overset{N_0}{\bullet} \cap P_A = \emptyset))$.
(11) Consider a transition $t \in \underline{cflat}(CA_c)$ with a label not used in cp. There is no

way to connect t to places in CA using $\underline{cflat}(CA'')$ because the label does not appear in the flow relation F^A. Hence, $(\forall\, t : t \in T_C \wedge \ell_1(t) \notin \alpha(N_B^W) : (\overset{N_1}{\bullet} t \cap P_A = \emptyset) \wedge (t \overset{N_1}{\bullet} \cap P_A = \emptyset))$.

(12) Similarly, one can show that $(\forall\, t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : (\overset{N_0}{\bullet} t \cap P_A = \overset{N_1}{\bullet} t' \cap P_A) \wedge (t \overset{N_0}{\bullet} \cap P_A = t' \overset{N_1}{\bullet} \cap P_A))$.

(13) P_B^W is activation safe in $(N_0, [i])$. This follows directly from the consistency of CA which is invariant under the replacements.

(14) $N_C^W = \underline{cflat}(CA_c) \leq_{pj} CS_c$, because c is consistent. $CS_c \leq_{pj} cp = N_B^W$, because $\overline{(C, \underline{cmap})}$ is consistent. Hence, $N_C^W \leq_{pj} N_B^W$.

Hence CA'' is a sound C-net in $\mathcal{C}$ and $\underline{cflat}(CA'') \leq_{pj} \underline{cflat}(CA)$. Since $\leq_{pj}$ is transitive, we conclude: $\underline{cflat}(CA'') \leq_{pj} \overline{CS_{c^t}}$. $\square$

Theorem 42 shows that a consistent system architecture is sound (i.e., no deadlocks, livelocks, or other anomalies) and that the actual behavior *conforms to the specification*. Moreover, the theorem also shows that it is possible to replace any consistent component by another consistent component which has an interface which is a subclass of the corresponding placeholder, i.e., the result can be used to effectively address *substitutability* issues!

Figure 4 shows the flattened system architecture introduced in Section 1. Since a system architecture composed of the components shown in figures 2 and 3 is consistent, the C-net shown in Figure 4 is guaranteed to be sound. Moreover, the C-net shown in Figure 4 is a subclass of the specification shown in Figure 2(a).

Also consider the system architecture composed of the components *coffee_machine*, *brewing_facility*, and *coin_handler* presented earlier. Since the system architecture is consistent, the actual behavior of the system conforms to the specification, i.e., the flattened system architecture is a subclass of the component specification shown in Figure 11. Moreover, the components *brewing_facility* and *coin_handler* can be replaced by other components satisfying a subclass/superclass relationship without jeopardizing the correct operation of the overall system!

5 Extensions based on other notions of inheritance

In Section 2.4, we mentioned the fact that in [4,5,10] four notions of inheritance have been identified. However, we introduced only one notion, i.e., projection inheritance, to avoid confusion. In this section we discuss potential future extensions of our framework based on other notions of inheritance.

Recall that the basic idea of projection inheritance can be characterized as follows.

If it is not possible to distinguish the behaviors of x and y when arbitrary methods of x are executed, but when only the effects of methods that are also present in y are considered, then x is a subclass of y.

For projection inheritance, all new methods (i.e., methods added in the subclass) are hidden using the abstraction operator τ_I. In a way all new methods are made internal such that the environment of the component cannot detect any differences. This means that a subclass under projection inheritance cannot offer any new functionality visible on the external interface of a component. Given this limitation, it is interesting to explore other notions of inheritance. Another basic form of inheritance is *protocol inheritance* which is based on blocking instead of abstraction.

If it is not possible to distinguish the external behavior of x and y when only methods of x that are also present in y are executed, then x is a subclass of y.

Intuitively, this alternative form of inheritance conforms to *blocking* calls to methods new in x. Component x is said to inherit the *protocol* of y if after blocking the new methods no environment can tell the difference between x and y. As long the new functionality is not "touched", the behavior of x and y is branching bisimilar. For a formal definition of protocol inheritance we refer to [4,5,10] where the encapsulation operator ∂_H is used to block a set of methods H. In Figure 7, N_1 and N_2 are subclasses of N_0 with respect to protocol inheritance.

The two mechanisms (i.e., blocking and hiding) result in two orthogonal inheritance notions. Therefore, we also consider combinations of the two mechanisms. A C-net is a subclass of another C-net under *protocol/projection inheritance* if and only if both by hiding the new methods and by blocking the new methods one cannot detect any differences, i.e., it is a subclass under both protocol and projection inheritance. In Figure 7, N_2 is the only subclass of N_0 with respect to protocol/projection inheritance. The two mechanisms can also be used to obtain a weaker form of inheritance. A component is a subclass of another component under *life-cycle inheritance* if and only if by blocking some newly added methods and by hiding some others one cannot distinguish between them. Life-cycle inheritance is more general than the other three inheritance relations. All C-nets shown in Figure 7 are subclasses of N_0 with respect to life-cycle inheritance. A detailed study of the four inheritance relations and the corresponding inheritance-preserving transformation rules can be found in [5,10].

At the moment, we are extending the framework to life-cycle inheritance. This extension will be realized as follows. In a systems architecture each component plugged into a placeholder is augmented with two lists of methods: one list for the methods that are hidden and another list for the methods that are blocked. The methods that are hidden are handled as described in this paper, i.e., these methods are made internal. The methods that are blocked are simply removed. In addition all inactive parts are also removed. Note that blocking one method on the interface of a component can deactivate large parts of the component. Given the proper requirements, we can generalize the main theorems presented in this paper, e.g., we can still prove that consistency of the system architecture implies soundness and conformance. Based on this result, it is easy to see that the framework can also be extended to protocol inheritance as being a special case of life-cycle inheritance. Moreover, the fourth notion of inheritance, protocol/projection inheritance, is a special case of the three other forms of inheritance. Therefore, the framework can also support this form of inheritance by simply restricting the notion of inheritance being used.

6 Related work

This paper presents a framework and results that build upon earlier results on WF-nets [1–3], inheritance [4,5,10], and a previous framework for component-based software architectures with Petri nets [20]. In [9] a software architecture is defined as the structure, which comprises software components, the externally visible properties of those components, and the relationships among them. A *good* architecture gives future flexibility: The possibility to evolve while maintaining the integrity and the *quality attributes* of a system. Quality attributes, also called architectural drivers, are used to measure the quality of an architecture. The quality attributes [9] of an architecture are performance, security, availability, functionality, usability, modifiability, portability, reusability, and integrability. Analysis and simulation of the architecture are used to determine them. Often these qualities compete and any design decision involves trade-offs. For instance between modifiability and performance or between scalability and reliability.

A general approach to develop architectures with their quality attributes is not available. Approaches to solve architectural problems are to categorize architectures and their particularities and to reuse reference models [25], to use patterns [12,14], or to use architectural blueprints [26]. But in most cases ad-hoc methods are used to solve architectural problems. The choice of the *Architecture Description Language* (ADL) often determines the ability to solve an architectural problem. In this section we relate the framework presented in this paper to other ADLs. In [29] an ADL is defined as a language that provides a concrete syntax and a conceptual framework for characterizing architectures. The conceptual framework typically subsumes the ADL's underlying semantic theory. Furthermore, in [29] a criterion which enables to determine whether or not a particular notation is an ADL is presented. An ADL must provide the means for the explicit specification of the building blocks of an architectural description: *components*, *connectors* and *architectural configurations*. Clearly the framework presented in this paper satisfies this criterion. First, components are first-class citizens of the framework (cf. Definition 31). Second, connectors between components can be identified in component architectures as a combination of (Petri net) places and arcs. The places and arcs connect the components in a component architecture. Third, it is obvious that for all component specifications multiple architectural configurations are allowed.

How does our framework relate to the numerous ADLs available on the market today? Examples are ARMANI, Rapide, Aesop, MetaH, UniCon, Darwin, Wright, C2 and SADL [15]. ADLs such as ARMANI, Rapide, Darwin, Wright, and Aesop typically view software architectures statically [28], i.e., analysis primarily focuses on syntactical and topological issues. Nevertheless, Darwin offers the possibility to execute "what if" scenarios and Rapide offers a constraint checker based on simulation. Another approach is the addition of process specifications to existing middleware technology, e.g., in [11] CORBA IDL's are extended with Petri nets to incorporate dynamic behavior. Several strategies to compare and to relate ADLs have been presented the last years. One strategy is by using the architecture interchange language ACME [16]. Its purpose is to capture the similarities of ADLs and to support the mapping of architectural specifications from one ADL to another. ACME is suited to do this at the syntactical level, but not (yet) at the semantic level. Another strategy to classify ADLs is by *architectural domains*, i.e., the problems or areas of concern that need to be addressed by ADLs [28]. The ADLs investigated in [28] are all supported by tools, which are tightly interwoven with the ADL. The framework presented in this paper has not (yet) been fully implemented in a tool. However Petri-net tools support the concepts of the framework. For the simulation of architectures Ex*Spect* [8] and other tools are available and for analysis Wo*fl*an [35] can be used. We will now discuss to what extent the framework is able to support the architectural domains defined in [28]. In the discussion we will separate the framework from the supporting toolset.

1. *Representation.* The framework provides a graphical notation of architectures by components and their methods in addition to the notation of labeled P/T-nets. By separating the specification from the architecture of a component, components are well understood among different stakeholders. For instance managers use the component specification to understand its behavior, while software engineers typically drill down in the hierarchy of the component architecture.

2. *Design Process Support.* There is no support on design decisions.

3.a. *Static analysis.* The component specifications and the flattened architectures of the framework are C-nets. The workflow analysis tool Wo*fl*an [35] can be used to perform correctness checks based on the properties defined in this paper. Definition 14 is an example of a static property that can be checked using Wo*fl*an.

3.b. *Dynamic analysis.* The framework has well defined semantics and therefore the behavior can be observed by executing the architecture. Petri-net simulation tools can be used to execute component specifications and flattened nets of architectures. The framework is extendable with color, time and priorities [19,22]. The addition of timing information enables to measure performance aspects of the architecture. The tool Ex*Spect* [8] can be used to analyze the performance of a given architecture using simulation. More sophisticated checks can be performed using Wo*fl*an [35]. Wo*fl*an 2.1 can analyze both soundness (Definition 16) and projection inheritance (Definition 26).

4.a. *Specification-Time evolution.* The subtyping mechanism used to support specification-time evolution is process inheritance [5]. This can be used to determine whether a component is a subclass of another component, and also whether the flattened net of an architecture is a subclass of the flattened net of another architecture.

4.b. *Execution-Time evolution.* The subtyping mechanisms mentioned in (4.a) allow to define migration rules for dynamic change. The rules are defined in [5] and can be used to migrate states between equivalent components.

5. *Refinement.* The refinement of framework architectures is supported by the hierarchy notion that is incorporated in the component definition.

6. *Traceability.* Traceability mechanisms between the various architectural views (implementation, process, control flow, data flow, graphical or textual) are not incorporated in the framework. However architectures specified in the framework are integrated models and it is possible to derive views from such models. For instance the tool Ex*Spect* [6,8] allows us to generate message sequence charts [21] from models; these are interaction scenarios between components.

7. *Simulation/Executability.* The dynamic behavior of the framework architectures can be simulated by Ex*Spect*. In Ex*Spect*, data elements can be added to the framework which allows for the observation of data transformations.

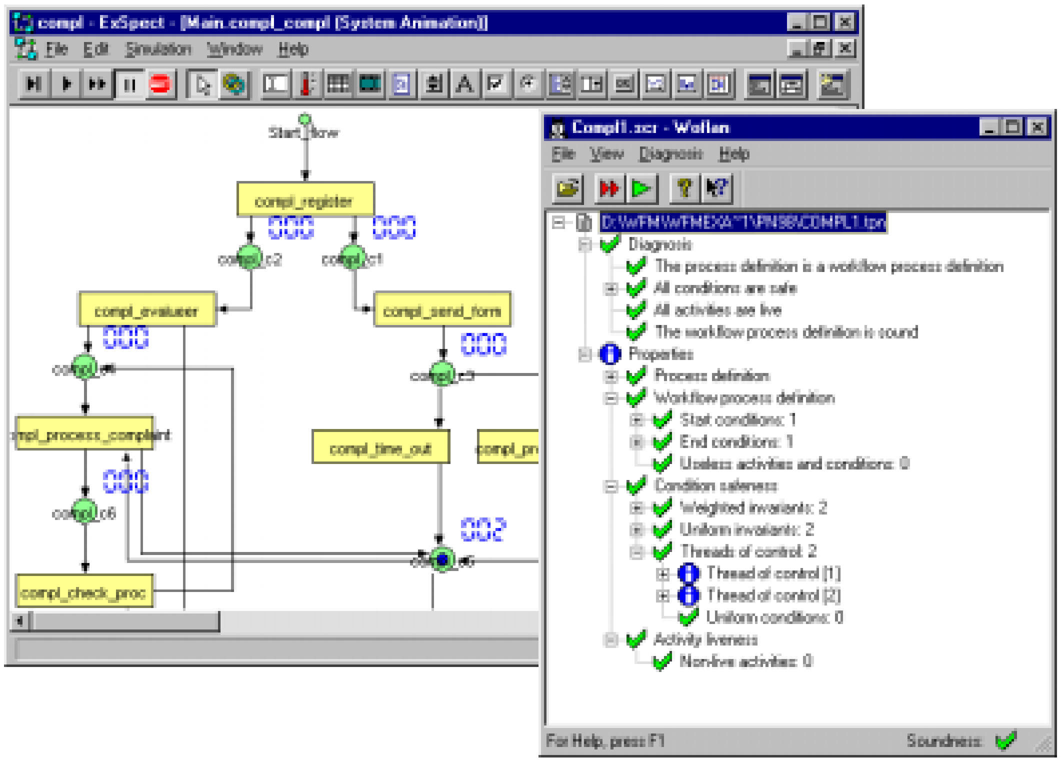
The framework introduced in this paper has a particular focus on the consistency of the dynamic behavior of components and architectures. It is not primarily intended to describe the data flow, the signature of method, or other aspects. But it enables software engineers to solve synchronization problems in complex architectures of distributed components that may have complex interaction scenarios. If we compare the framework with other ADLs, then we see that it still has a broad focus (it addresses almost all the architectural domains in the list). Nevertheless, the framework provides particular strong results with respect to refinement and evolution of architectures. Key to these results are the inheritance-preserving transformation rules which preserve the behavior of a component. In the last decade several researchers [7,23,24,27] explored notions of behavioral inheritance (also named subtyping or substitutability). Researchers in the domain of formal process models (e.g., Petri nets and process algebras) have tackled similar questions based on the explicit representation of a process by using various notions of (bi)simulation [10]. The inheritance notion used in this paper is characterized by the fact that it is equipped with both *inheritance-preserving transformation rules* to *construct* subclasses (see Section 2.4 and [4,10]) and *transfer rules* to *migrate* instances from a superclass to a subclass and vice versa [5]. These features are relevant for a both constructive and robust approach towards truly component-based software development.

7 Conclusion

In this paper we introduced a framework to model software architectures. Like any framework for software architectures the interaction amongst components is emphasized. Unlike most frameworks the scope is restricted to the dynamics of software architectures, i.e., we abstract from data and other relevant perspectives. This restriction is motivated by the fact that the dynamic behavior of components in a software architecture is often ignored or described at a level which defies formal analysis. We think it is important to incorporate the dynamic behavior of components as first-class citizens in an architectural framework: Concurrency issues are very important for the design of large software systems and should not be ignored. Clearly, it is not possible to do formal analysis on real-life systems if all perspectives (including data) are incorporated. Therefore, we choose to abstract from non-behavioral aspects.

The framework presented in this paper has been used to address one of the key issues of component-based software development: *consistency*. We have defined consistency at the level of a single component and at the level of a system architecture. Clearly consistency is very important in the context of component-based software development: Will a component "fit" or not? To answer this question, we put the notion of projection inheritance [4,10] to work. Projection inheritance can be used to check whether a component actually provides the external behavior required. The inheritance notion is equipped with concrete inheritance-preserving design patterns and allows for modular conformance testing of the system architecture. One of the main results of the paper is Theorem 40 which shows that projection inheritance is compositional. Based on the compositionality of projection inheritance, we can prove that consistency implies the correct behavior of the overall system, i.e., the system is free of deadlocks and other anomalies and realizes the specification of the top-level component.

In the future, we plan to extend our framework with other notions of inheritance. The three other notions of inheritance presented in [4,10] can also be used to obtain complementary compositionality results. For example, the notion of *protocol inheritance* [4,10], which is based on encapsulation rather than abstraction of methods, allows for very generic components whose functionality is only partly used in a given context. Another extension of our framework is the dynamic replacement of components using the transfer rules presented in [5]. The transfer rules allow for the on-the-fly migration of execution states from one component to another as long as there is a subclass/superclass relationship. We also plan to work on the extension with data and methods signatures. For example, it would be interesting to extend the work presented in [11] with our notion of inheritance. Finally, we plan to adapt our tools Wo*fl*an [35] and Ex*Spect* [8] to serve the framework presented in this paper. Wo*fl*an can be used to check the requirements involving soundness and consistency, i.e., Wo*fl*an 2.1 can verify soundness and projection inheritance. Ex*Spect*



```
D:\WFM\WFMEXA~1\PN98\COMPL1.tpn
  Diagnosis
    The process definition is a workflow process definition
    All conditions are safe
    All activities are live
    The workflow process definition is sound
  Properties
    Process definition
    Workflow process definition
      Start conditions: 1
      End conditions: 1
      Useless activities and conditions: 0
    Condition safeness
      Weighted invariants: 2
      Uniform invariants: 2
      Threads of control: 2
        Thread of control [1]
        Thread of control [2]
        Uniform conditions: 0
    Activity liveness
      Non-live activities: 0
```

Fig. 15. A screenshot of Wo*fl*an (front) and Ex*Spect* (back): Wo*fl*an can be used to check consistency and Ex*Spect* can be used to prototype software architectures.

can be used as a prototyping environment for experimenting with component-based software architectures. Both tools have been developed under the supervision of the first two authors and are illustrated in Figure 15.

Acknowledgements The authors would like to thank Twan Basten for his excellent work on inheritance of dynamic behavior and Eric Verbeek for the development of Wo*fl*an.

References

[1] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.

[2] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[3] W.M.P. van der Aalst. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin, 2000.

[4] W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997.

[5] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 2000 (to appear).

[6] W.M.P. van der Aalst, P. de Crom, R. Goverde, K.M. van Hee, W. Hofman, H. Reijers, and R.A. van der Toorn. ExSpect 6.4: An Executable Specification Tool for Hierarchical Colored Petri Nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 455–464. Springer-Verlag, Berlin, 2000.

[7] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundation of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, Berlin, 1991.

[8] Deloitte & Touche Bakkenist. ExSpect Home Page. http://www.exspect.com.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 1998.

[10] T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998.

[11] R. Bastide and P. Palanque et al. Petri-Net Based Behavioural Specification of CORBA Systems. In *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 66–85. Springer-Verlag, Berlin, 1999.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A system of Patterns*. John Wiley and Sons, New York, 1996.

[13] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.

[15] D. Garlan, R.T. Monroe, and D. Wile. ADL's and Related Languages, Carnegie Mellon. http://www.cs.cmu.edu/˜acme/adltk/adls.html.

[16] D. Garlan, R.T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[17] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[18] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[19] K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.

[20] K.M. van Hee, R.A. van der Toorn, J. van der Woude, and P. Verkoulen. A Framework for Component Based Software Architectures. In W.M.P. van der Aalst, J. Desel, and R. Kaschek, editors, *Software Architectures for Business Process Management (SABPM'99)*, pages 1–20, Heidelberg, Germany, June 1999. Forschungsbericht Nr. 390, University of Karlsruhe, Institut AIFB, Karlsruhe, Germany.

[21] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96). Technical report, ITU-TS, Geneva, 1996.

[22] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1996.

[23] H. Kilov and W. Harvey, editors. *Object-Oriented Behavioral Specifications*, volume 371 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, USA, 1996.

[24] H. Kilov, B. Rumpe, and I. Simmonds, editors. *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, USA, 1999.

[25] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architecture Styles. In *Software Architecture, Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 225–243, San Antonio, TX, 1999.

[26] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[27] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[28] N. Medvidovic and D. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages . In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 199–212, Santa Barbara, October 1997.

[29] N. Medvidovic and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 60–67, Zurich, Switzerland, 1997.

[30] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[31] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[32] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/otl/technical/umlrt.html.

[33] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[34] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[35] E. Verbeek and W.M.P. van der Aalst. Woflan Home Page, Eindhoven University of Technology, Eindhoven, The Netherlands. http://www.win.tue.nl/ ˜woflan.