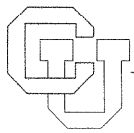
Component-Based Software Architectures: A Framework Based on Inheritance Behavior

W.M.P van der Aalst
K.M. van Hee
R.A. van der Toorn

CU-CS-892-99



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

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

W.M.P. van der Aalst[1,3], K.M. van Hee[1,2], and R.A. van der Toorn[1,2]

[1] Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. wsinwa@win.tue.nl
[2] Deloitte & Touche Bakkenist, P.O. Box 23103, NL-1100 DP Amsterdam, The Netherlands. kvhee@bakkenist.nl, rvdtoorn@bakkenist.nl
[3] 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 interconnected 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 focussing 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.

1 Introduction

Research in the domain of *component-based software architectures* [21, 22] 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 Definition Languages* (ADLs), e.g., ARMANI, Rapide, Darwin, Wright, and Aesop, that have been proposed (cf. [17]). Another example is the extension of UML based on the ROOM language [20] 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) [22].

In this paper, we focus on the *dynamic behavior* of components rather than the passing of data, the signature of methods, and naming issues. Therefore, we use Petri nets [19] to describe the interfaces between components. Figure 1 illustrates the notion of component we will use throughout this paper. A component has a *Name* and a *Component Specification* (CS). The CS 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 CA may refer to other components by using *Component Placeholders* (CPs). Every CP describes the functionality of a component used in the CA in terms of a C-net. A component is *atomic* if it contains no other components, i.e., there are no CPs in its architecture. A *System Architecure* (SA) is a set of interconnected components, i.e., CPs are linked to concrete components.

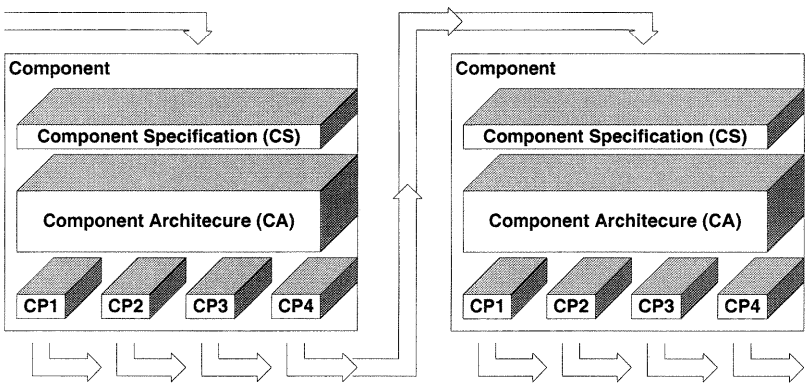


Fig. 1. A component consists of a component interface, a component architecture, and component placeholders.

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 CS. A SA is consistent if its components are consistent and every CP is mapped onto a component which actually provides the functionality specified in the CP. This paper uses the notion of *projection inheritance* [4, 8] 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 SA. 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 remainder of the paper is organized as follows. First, we introduce the notions this works 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 SA 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 [10, 18, 19]. 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:*

1. $P \subseteq U$ *is a finite set of* places,
2. $T \subseteq U$ *is a finite set of* transitions *such that* $P \cap T = \emptyset$,
3. $M \subseteq L_v$ *is a finite set of* methods *such that* $M \cap (P \cup T) = \emptyset$,
4. $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation, *and*
5. $\ell : T \rightarrow M \cup \{\tau\}$ *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: ${}^{N}\bullet x$ is the preset of node x in net N and $x{}^{N}\bullet$ 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 is 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 iff 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. 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* $P' = P \backslash \{\underline{source}(N), \underline{sink}(N)\}$ *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:*[1]

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

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 execution sequence activating this transition.

Theorem 1 (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 given C-net.

Lemma 1. *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]. □

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

Definition 17 (Alphabet operator α). *Let* (N, s) *be a marked, labeled P/T-net in* $\mathcal{N}$, *with* $N = (P, T, M, F, \ell)$. $\alpha : \mathcal{N} \rightarrow \mathcal{P}(L_v)$ *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* [11] 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 18. *The class of marked, labeled P/T-nets $\mathcal{N}$ is equipped with the following termination predicate:* $\downarrow = \{(N,[o]) \mid N \text{ is a C-net} \wedge o = \underline{sink}(N)\}$.

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 19. *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 20. *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 21 (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 $p\mathcal{R}q$.*

Figure 2 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

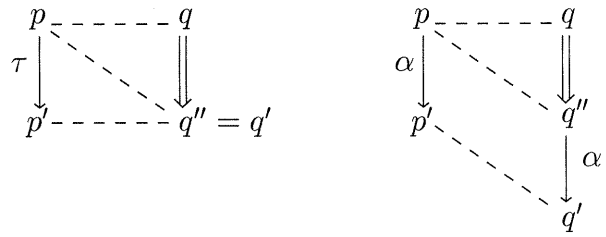


Fig. 2. The essence of a branching bisimulation.

not simulate. The third property in Definition 21 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 [8] for more details and pointers to other notions of branching bisimilarity.

2.4 Inheritance

In [4, 5, 8] 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 5). 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 22 (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 23 (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 $(\tau_I(N_1), [i]) \sim_b (N_0, [i])$.*

Based on this notion of 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 proof we summarize some of the results given in [4, 5, 8].

Theorem 2 (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 $N_1 \leq_{pp} N_0$.

Theorem 3 (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* ${}^N\bullet t_p = \{p\}$,
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 $N_1 \leq_{pj} N_0$.

Theorem 4 (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. ${}^N\bullet p = \{t_o\}$, $p^{N}\bullet = \{t_i\}$,
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 $N_1 \leq_{pj} N_0$.

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, 8] 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 24 (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) *$\ell^A : T^A \cup B \not\to M^S \cup \{\tau\}$ 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 atomic components $C^A = \emptyset$. For non-atomic components the architecture contains a set of placeholders C^A. The placeholders are used for *plugging in* other components. Therefore, 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.

Figure 3 shows an example of a component which 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

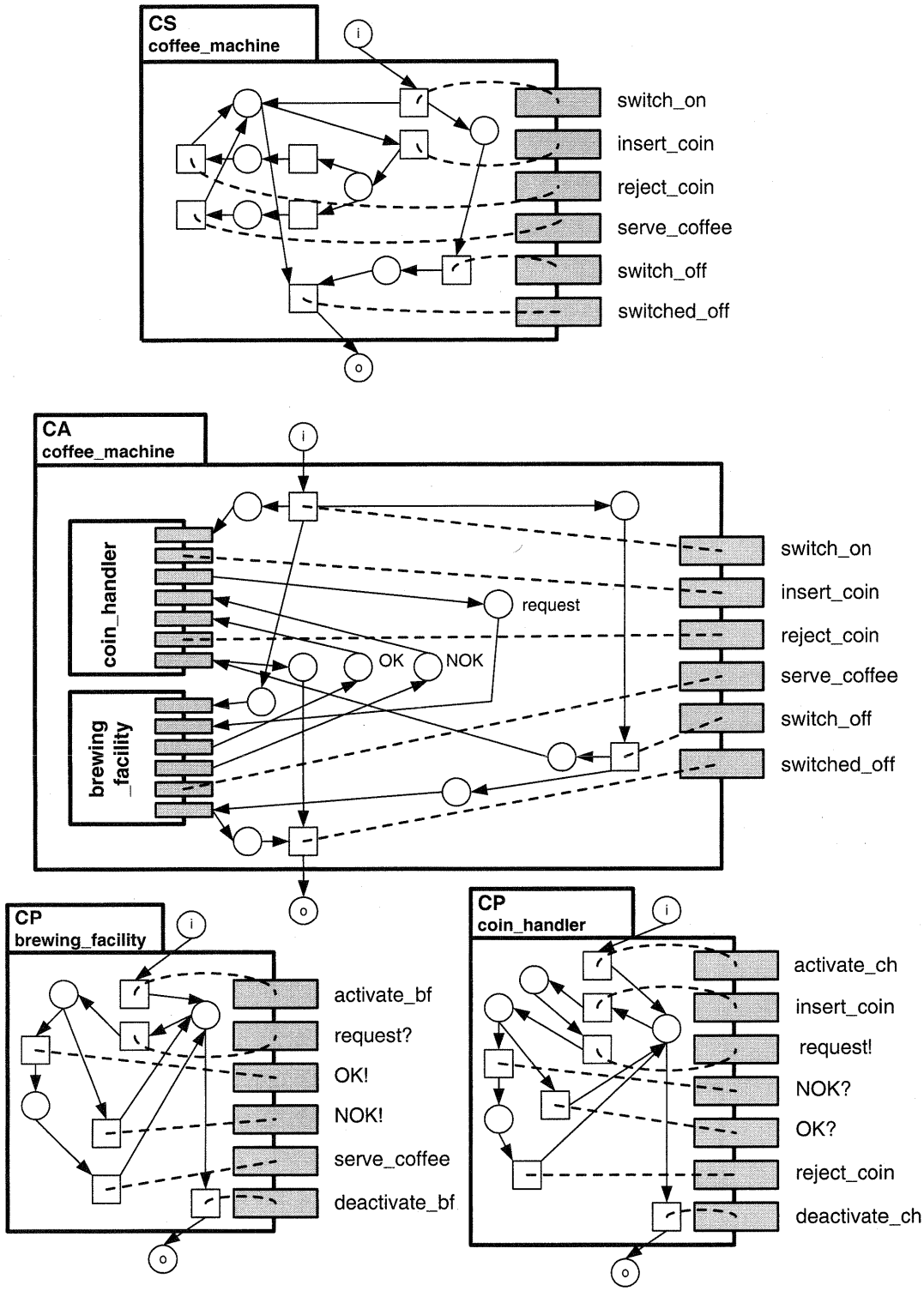


Fig. 3. The component *coffee_machine*.

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 3. 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.

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 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 25 (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. $M = rng(\ell) \backslash \{\tau\}$.

Definition 26 (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 $N = \underline{cflat}(CA)$. (CS, CA) is consistent if and only if*

1. $M^S = rng(\ell^S)\backslash\{\tau\}$,
2. *for every* $cp \in C^A$: $M^{SA}_{cp} = rng(\ell^{SA}_{cp})\backslash\{\tau\}$,
3. $M^S = (\{\ell^A(t) \mid t \in T^A\} \cup \bigcup_{cp \in C^A}\{\ell^A(cp,l) \mid l \in M^{SA}_{cp}\})\backslash\{\tau\}$,
4. N *is a sound C-net, i.e.,* $N \in \mathcal{C}$,
5. *for any* $cp \in C^A$, $t, t' \in \underline{start}(cp)$: ${}^N\bullet t = {}^N\bullet t'$,
6. *for any* $cp \in C^A$, $t, t' \in \underline{stop}(cp)$: $t^N\bullet = t'^N\bullet$,
7. *for any* $cp \in C^A$, $t \in T^{SA}_{cp}$, *and* $t' \in \underline{start}(cp)$: *all non-trivial directed paths from* t *to* t' *in* N *contain at least one occurrence of a transition in* $\underline{stop}(cp)$, *and*
8. $N \leq_{pj} CS$.

Definition 26 gives the minimal set of requirements any component should satisfy. The first three requirements state that the methods offered at the various levels should actually be present. The flattened architecture, i.e., the functionality guaranteed by the architecture provided the correct operation of subcomponents, is sound. Subcomponents are started and stopped correctly. A subcomponent is not allowed to be able to activate itself. Therefore, paths from inside a component to the start transitions of the component are excluded. Note that after terminating the subcomponent it may be activated again. Finally, we require that the flattened architecture is a subclass of the component specification with respect to projection inheritance.

The component shown in Figure 3 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 4 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 26 we can derive the following properties.

Lemma 2. *For any consistent component* (CS, CA) *with* $CS = (P^S, T^S, M^S, F^S, \ell^S)$ *and* $\underline{cflat}(CA) = (P, T, C, F, \ell)$: $\{\ell^S(t) \mid t \in \underline{start}(CS)\} = \{\ell(t) \mid t \in \underline{start}(\underline{cflat}(CA))\}$ *and* $\{\ell^S(t) \mid t \in \underline{stop}(CS)\} = \{\ell(t) \mid t \in \underline{stop}(\underline{cflat}(CA))\}$

Proof. The construction of $\underline{cflat}(CA)$ guarantees that no new labels are introduced. Combining this with $\underline{cflat}(CA) \leq_{pj} CS$ implies that the behavior of CS and $\underline{cflat}(CA)$ should match with respect to the visible steps. Since both CS and $\underline{cflat}(CA)$ are C-nets, they always start (end) with a visible step. Hence the lemma holds. □

Lemma 3. *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* $\{t \in T^A \mid (t,i) \in F^A\} \cup \{(cp,l) \in$

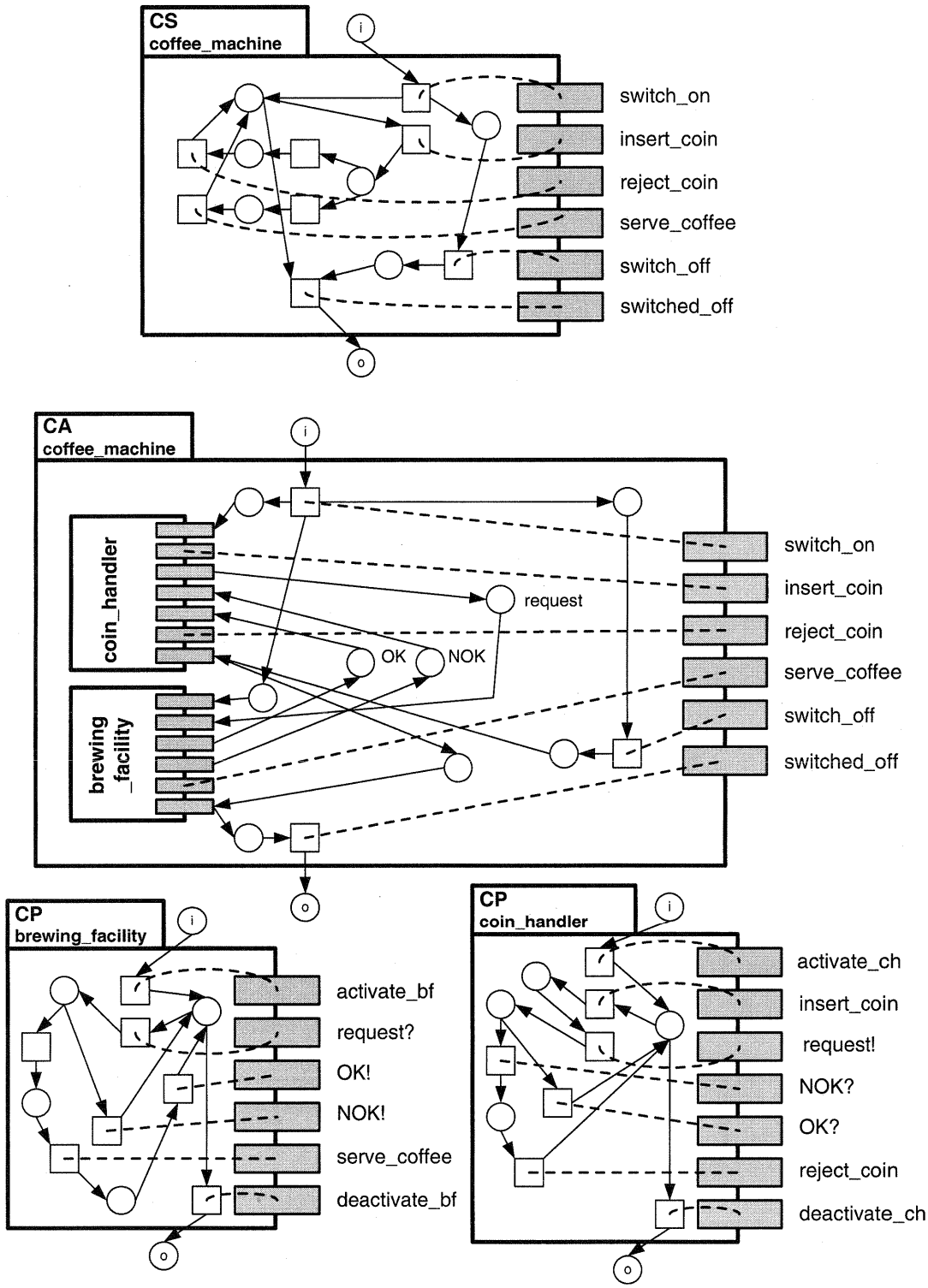


Fig. 4. A consistent version of the *coffee_machine* component: The two subcomponents are deactivated sequentially and coffee is served before the acknowledgement is sent.

$C^A \times L_v \mid ((cp,l),i) \in F^A\} = \emptyset$ *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. Similarly, it can be shown that there is precisely one source 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.

A *system architecture* consists of a set of components where components are plugged into placeholders of other components.

Definition 27 (System architecture). *Let* C *be 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 28 (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,*[2]

Let us consider the system architecture for a coffee machine. The component shown in Figure 4 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 5 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 6). Note that both subcomponents are atomic, i.e., the system architecture for a coffee machine has two levels and comprises three components. Clearly, this simple system architecture is well-formed.

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

Definition 29 (Consistent). *Let* $(C, \underline{cmap})$ *be a well-formed system architecture such that for any* $c \in C$: $c = (CS_c, CA_c)$, $CS_c = (P_c^S, T_c^S, M_c^S, F_c^S, \ell_c^S)$, *and* $CA_c = (P_c^A, T_c^A, C_c^A, F_c^A, \ell_c^A)$. $(C, \underline{cmap})$ *is* consistent *if and only if*

[2] 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.

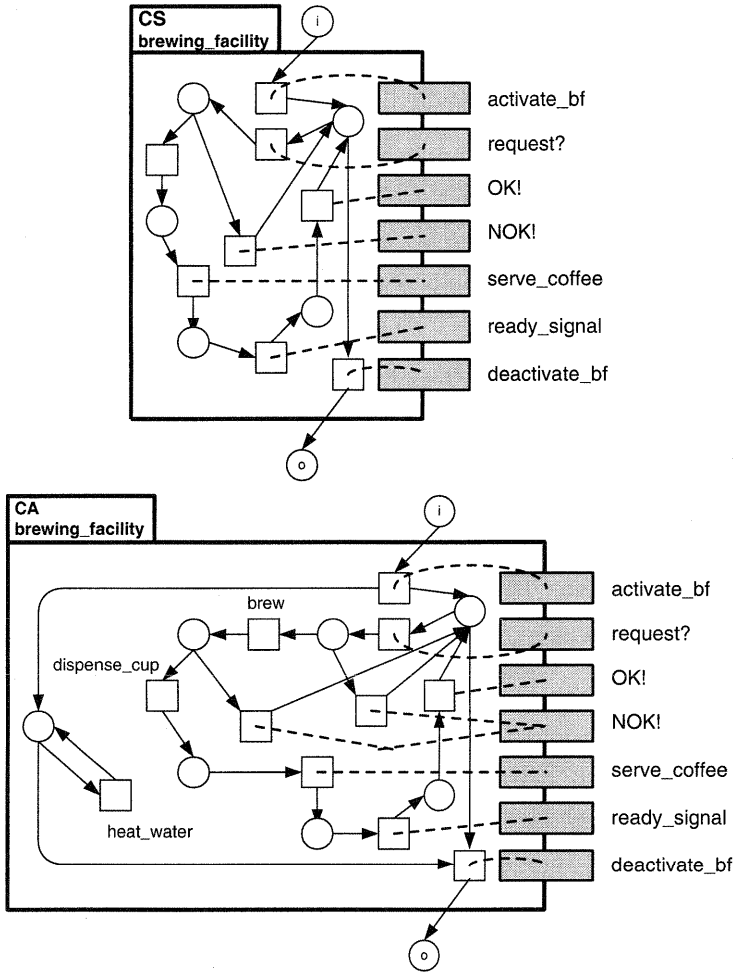


Fig. 5. The component *brewing_facility*.

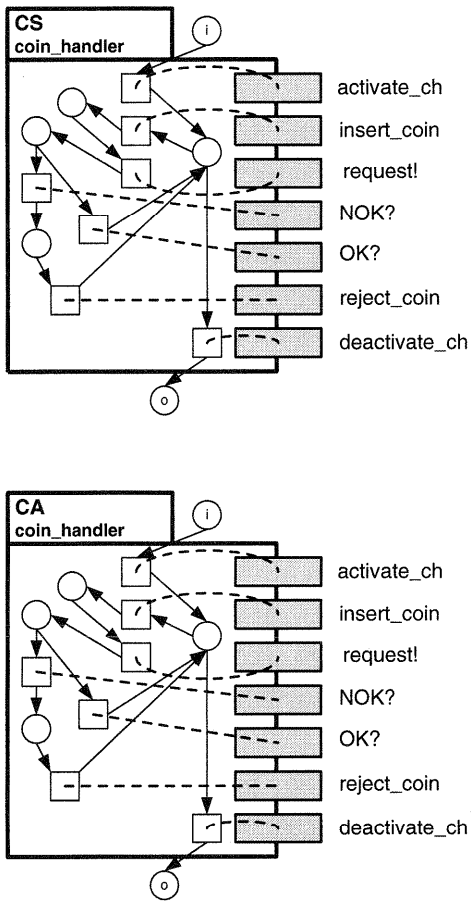


Fig. 6. The component *coin_handler*.

1. *each component* $c \in C$ *is consistent, and*
2. *for all* $c \in C$, $c' \in C$, *and* $cp \in C_c^A$ *such that* $\underline{cmap}(c, cp) = c'$ *and* $cp = (P_{cp}^{SA}, T_{cp}^{SA}, M_{cp}^{SA}, F_{cp}^{SA}, \ell_{cp}^{SA})$*:*
   (a) $CS_{c'} \leq_{pj} cp$,
   (b) $\{\ell_{c'}^{S}(t) \mid t \in \underline{start}(CS_{c'})\} = \{\ell_{cp}^{SA}(t) \mid t \in \underline{start}(cp)\}$, *and*
   (c) $\{\ell_{c'}^{S}(t) \mid t \in \underline{stop}(CS_{c'})\} = \{\ell_{cp}^{SA}(t) \mid t \in \underline{stop}(cp)\}$.

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 there should be a match between the methods used 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 4, the component *brewing_facility* shown in Figure 5, and the component *coin_handler* shown in Figure 6. 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 4. 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.

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. For this purpose we first formulate and prove a rather general theorem which addresses the notion of compositionality in the context of projection inheritance.

Theorem 5 (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. $(\forall t, t' : t \in \underline{start}(N_B^W) \wedge t' \in \underline{start}(N_C^W) : {}^{N_0}\bullet t = {}^{N_1}\bullet t')$, *i.e., start transitions have identical sets of input places.*
10. $(\forall t, t' : t \in \underline{stop}(N_B^W) \wedge t' \in \underline{stop}(N_C^W) : t^{N_0}\bullet = t'^{N_1}\bullet)$, *i.e., stop transitions have identical sets of output places,*
11. $(\forall t : t \in T_B \wedge \ell_B(t) = \tau : ({}^{N_0}\bullet t \cap P_A = \emptyset) \wedge (t^{N_0}\bullet \cap P_A = \emptyset))$,
12. $(\forall t : t \in T_C \wedge \ell_1(t) \notin \alpha(N_B^W) : ({}^{N_1}\bullet t \cap P_A = \emptyset) \wedge (t^{N_1}\bullet \cap P_A = \emptyset))$,
13. $(\forall t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : ({}^{N_0}\bullet t \cap P_A = {}^{N_1}\bullet t' \cap P_A) \wedge (t^{N_0}\bullet \cap P_A = t'^{N_1}\bullet \cap P_A))$,
14. $(\forall t, t' : t \in T_B \wedge t' \in \underline{start}(N_B^W)$: *all non-trivial directed paths in* N_0 *from* t *to* t' *contain at least one occurrence of a transition in* $\underline{stop}(N_B^W)$), *and*
15. $N_C^W \leq_{pj} N_B^W$,

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 crucial to 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 $I = \alpha(N_C^W) \backslash \alpha(N_B^W) = \alpha(N_1) \backslash \alpha(N_0)$.
   ◇ This follows directly from the definition of projection inheritance.
2. $(\forall t, t' : t \in T_B \wedge t' \in T_B \wedge \ell_B(t) = \ell_B(t') : ({}^{N_0}\bullet t \cap P_A = {}^{N_0}\bullet t' \cap P_A) \wedge (t^{N_0}\bullet \cap P_A = t'^{N_0}\bullet \cap P_A))$, i.e., transitions in T_B with identical labels have identical effects on the interface $P_A \cap P_B$.
   ◇ 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.

3. $(\forall t,t' : t \in T_C \wedge t' \in T_C \wedge \ell_C(t) = \ell_C(t') : ({}^{N_1}\bullet t \cap P_A = {}^{N_1}\bullet t' \cap P_A) \wedge (t^{N_1}\bullet \cap P_A = t'^{N_1}\bullet \cap P_A))$, i.e., transitions in T_C with identical labels have identical effects on the interface $P_A \cap P_C$.
   ◇ 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.
4. $(\forall t,t' : t \in \underline{start}(N_B^W) \wedge t' \in \underline{start}(N_B^W) : {}^{N_0}\bullet t = {}^{N_0}\bullet t')$, $(\forall t,t' : t \in \underline{stop}(N_B^W) \wedge t' \in \underline{stop}(N_B^W) : t^{N_0}\bullet = t'^{N_0}\bullet)$, $(\forall t,t' : t \in \underline{start}(N_C^W) \wedge t' \in \underline{start}(N_C^W) : {}^{N_1}\bullet t = {}^{N_1}\bullet t')$, $(\forall t,t' : t \in \underline{stop}(N_C^W) \wedge t' \in \underline{stop}(N_C^W) : t^{N_1}\bullet = t'^{N_1}\bullet)$.
   ◇ This follows directly from the requirement that start/stop transitions in different nets have identical sets of input/output places.
5. N_0, N_1, N_B^W, and N_C^W completely determine N_A, N_B, and N_C.
   ◇ $N_A = N_0 \cap N_1$, $N_B = ({}^{N_0}\bullet T_B^W \cup T_B^W {}^{N_0}\bullet, T_B^W, M_B^W, F_0 \cap ((P_B \times T_B) \cup (T_B \times P_B)), \ell_B^W)$, and $N_C = ({}^{N_1}\bullet T_C^W \cup T_C^W {}^{N_1}\bullet, T_C^W, M_C^W, F_1 \cap ((P_C \times T_C) \cup (T_C \times P_C)), \ell_C^W)$.
6. For any $s_0 \in [N_0,[i]\rangle$, $t \in \underline{start}(N_B^W)$, and $p \in P_B \backslash P_A$: if $(N_0,s_0)[t\rangle$, then place p is empty in s_0.
   ◇ This property is crucial and depends heavily on the safeness of the input places of $\underline{start}(N_B^W)$ in $(N_0,[i])$ and the requirement that all non-trivial directed paths in N_0 from a transition inside N_B to one of the start transitions in N_B contain at least one of the stop transitions in N_B. More details are given below.
7. 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 $s_B \in [N_B^W,[i_B]\rangle$.
   ◇ 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, the previous property clearly shows that this is only possible if each place in $P_B \backslash P_A = P_0 \backslash P_A$ is empty.
8. $(\forall t,t' : t \in T_C \wedge t' \in \underline{start}(N_C^W)$: all non-trivial directed paths in N_1 from t to t' contain at least one occurrence of a transition in $\underline{stop}(N_C^W))$.
   ◇ The connections of transitions in N_C are identical to the connections of the transitions in N_B with respect to the interface $P_A \cup P_B$. Therefore, similar to N_0, there are no such paths.
9. For any $s_1 \in [N_1,[i]\rangle$, $t \in \underline{start}(N_C^W)$, and $p \in P_C \backslash P_A$: if $(N_1,s_1)[t\rangle$, then place p is empty.
   ◇ Similar arguments apply. Again the safeness of the input places of $\underline{start}(N_C^W)$ and the requirement that all non-trivial directed paths in N_1 from a transition inside N_C to one of the start transitions in N_C contain at least one of the stop transitions in N_C are crucial.
10. Any marking $s_1 \in [N_1,[i]\rangle$ can be partitioned into s_A and s_C such that $s_1 = s_A + s_C$, $s_A \in \mathcal{B}(P_A)$, $s_C \in \mathcal{B}(P_1 \backslash P_A)$, and $s_C = \mathbf{0}$ or $s_C \in [N_C^W,[i_C]\rangle$.
    ◇ Since $P_A \cap P_B = P_A \cap P_C$ the same arguments apply.

The first five observations are straightforward. The other observations are more involved. Therefore, we show in more detail that for any $s_0 \in [N_0, [i]\rangle$, $t \in \underline{start}(N_B^W)$, and $p \in P_B \backslash P_A$: if $(N_0, s_0)[t\rangle$, then place p is empty. For this purpose, we use proof by contradiction, i.e., we assume that there is a firing sequence σ such that $(N_0, [i])\ [\sigma\rangle\ (N_0, s_0)$, $t \in \underline{start}(N_B^W)$, $(N_0, s_0)[t\rangle$, and $p \in P_B \backslash P_A$ is marked in s_0. Without loss of generality, we further assume that s_0 was the first state in the sequence having these properties (i.e., a start transition is enabled while a place in $P_B \backslash P_A$ 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 $\underline{stop}(N_B^W)$, i.e., σ_1 is either empty or ends with the last firing of a transition in $\underline{stop}(N_B^W)$. The first sequence ends in state s' (i.e., $(N_0, [i])\ [\sigma_1\rangle\ (N_0, s')$). Note that in s' all places in $P_B \backslash P_A$ are empty. (Otherwise there would have been a prefix of σ containing the anomaly.) Now we concentrate on the second subsequence: $(N_0, s')\ [\sigma_2\rangle\ (N_0, s_0)$. In this sequence no transition in $\underline{stop}(N_B^W)$ fires. Therefore, we remove all transitions $\underline{stop}(N_B^W)$ from N_0 and name the new net N. Note that $(N, s')\ [\sigma_2\rangle\ (N, s_0)$. The requirement that all non-trivial directed paths in N_0 from a transition inside N_B to one of the start transitions in N_B contain at least one of the stop transitions in N_B implies that we can partition the transitions of N in two subsets T_X and T_Y such that $\{t \in T_A \mid t {}^{N}\!\bullet \cap {}^{N}\!\bullet \underline{start}(N_B^W) \neq \emptyset\} \subseteq T_X$, $T_B \subseteq T_Y$, and ${}^{N}\!\bullet T_X \cap T_Y {}^{N}\!\bullet = \emptyset$ because all stop transitions have been removed. Now we apply the well-known exchange lemma (see for example page 23 in [10]) which allows us to project σ_2 onto the transitions in T_X and T_Y: σ_{2X} and σ_{2Y}. Since ${}^{N}\!\bullet T_X \cap T_Y {}^{N}\!\bullet = \emptyset$, 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'' each of the input places of the start transitions of N_B contains multiple tokens. (Note that σ_{2Y} marks a place in $P_B \backslash P_A$, i.e., fires at least one start transition of N_B, and also enables a start transition of N_B 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 $(N_0, [i])$. Therefore, $(N_0, [i])$ cannot be a sound C-net and we find a contradiction.

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: $\mathcal{R}_{01} = \{((N_0, s_A + s_B), \tau_I(N_1, s_A + s_C)) \mid s_A \in \mathcal{B}(P_A) \wedge s_B \in \mathcal{B}(P_0 \backslash P_A) \wedge s_C \in \mathcal{B}(P_1 \backslash P_A) \wedge s_A + s_B \in [N_0, [i]\rangle \wedge s_A + s_C \in [N_1, [i]\rangle \wedge ((s_B = \mathbf{0} \wedge s_C = \mathbf{0}) \vee ((N_B^W, s_B) \mathcal{R}_{BC} \tau_I(N_C^W, s_C)))\}$.

The remainder proof consists of two parts. In the first part, it is shown that $\mathcal{R}_{01}$ is a branching bisimulation and that $(N_0, [i]) \mathcal{R}_{01} \tau_I(N_1, [i])$. In the second part, it is shown that N_1 is a sound C-net.

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 ${}^{N_0}\bullet t \cup t^{N_0}\bullet = {}^{N_1}\bullet t \cup t^{N_1}\bullet \subseteq P_A$. 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 $t \in T_B$.
     - If $s_B = \mathbf{0}$ and $s_C = \mathbf{0}$, then $t \in \underline{start}(N_B^W)$. Hence, each place in ${}^{N_0}\bullet t$ 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 $\ell_0(t) = \ell_1(t')$, ${}^{N_1}\bullet t' = {}^{N_0}\bullet t \subseteq \mathcal{B}(P_A)$. 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 $(N_1, s_1) \Longrightarrow (N_1, s_1'')\ [(\ell_0(t))\rangle\ (N_1, s_1') \wedge (N_0, s_0)\mathcal{R}_{01}(N_1, s_1'') \wedge (N_0, s_0')\mathcal{R}_{01}(N_1, s_1')$.
     - If $s_B \neq \mathbf{0}$ or $s_C \neq \mathbf{0}$, then $(N_B^W, s_B)\mathcal{R}_{BC}(\tau_I(N_C^W, s_C))$. Since $\mathcal{R}_{BC}$ is a branching bisimulation, $(N_B^W, [i_B])\mathcal{R}_{BC}\tau_I(N_C^W, [i_C])$, $s_B \in [N_B^W, [i_B]\rangle$, and $s_C \in [N_C^W, [i_C]\rangle$, it is straightforward to show that a sequence consisting of zero or more silent steps and a step similar to t can be executed in (N_1, s_1). Note that it is essential that the effects of all non-τ steps are identical with respect to the places in P_A, i.e., $(\forall t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : ({}^{N_0}\bullet t \cap P_A = {}^{N_1}\bullet t' \cap P_A) \wedge (t^{N_0}\bullet \cap P_A = t'^{N_1}\bullet \cap P_A))$. 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 identical to the proof in the other direction.
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 $(N_B^W, \mathbf{0})\mathcal{R}_{BC}\tau_I(N_C^W, s_C)$ which is not possible because from s_C it is possible to fire a non-τ-labeled transition, i.e., a transition in $\underline{stop}(N_B^W)$. 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

Remains to prove that N_1 is a sound C-net. It is easy to see that N_1 is a WF-net: There is one source place i, one source 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 $s_C \in [N_C^W, [i_C]\rangle$ such that for any place $p \in P_1 \backslash P_A$: $s_1(p) = s_C(p) \leq 1$.
- 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 $P_1 \backslash P_A$ are also empty, because otherwise there would be a nonempty bag s_C such that $s_C \neq [o_B]$ and $(N_B^W, \mathbf{0})\mathcal{R}_{BC}$ $\tau_I(N_C^W, s_C)$. 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 $T_1 \backslash T_A$ are not dead, because there are no dead transitions in $(N_C^W, [i_C])$.

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

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 *aflat* to translate a system architecture into a labeled P/T net.

Definition 30 (Flattened architecture). *Let $(C, cmap)$ 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 $aflat(C, cmap)$ 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 28.*
$CA = (P^A, T^A, C^A, F^A, \ell^A) := CA_{c^t}$
$hmap(cp) := cmap(c^t, cp)$ *for all* $cp \in C_{c^t}^A$

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

Step 3 *Select a $cp \in C^A$.*
$c := hmap(cp)$
$CA' = (P^{A'}, T^{A'}, C^{A'}, F^{A'}, \ell^{A'}) := strip(CA_c)$
$P^{A''} := P^A \cup P^{A'}$
$T^{A''} := T^A \cup T^{A'}$
$C^{A''} := (C^A \backslash \{cp\}) \cup C_c^A$
$F^{A''} := (F^A \backslash ((\{cp\} \times L_v) \times P^A) \cup (P^A \times (\{cp\} \times L_v))) \cup F^{A'} \cup \{(p, x) \in P^A \times dom(\ell^{A'}) \mid (p, (cp, \ell^{A'}(x))) \in F^A\} \cup \{(x, p) \in dom(\ell^{A'}) \times P^A \mid ((cp, \ell^{A'}(x)), p) \in F^A\}$
$dom(\ell^{A''}) := (dom(\ell^A) \backslash (\{cp\} \times L)) \cup dom(\ell^{A'})$.
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 atomic components. 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 [12]. The following theorem uses the compositionality result of Theorem 5 to show that consistency implies the proper operation of the whole system.

Theorem 6 (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 30 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 26).

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 5. 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 5:

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 $(\forall t, t' : t \in \underline{start}(N_B^W) \wedge t' \in \underline{start}(N_C^W) : {}^{N_0}\bullet t = {}^{N_1}\bullet t')$, i.e., start transitions have identical sets of input places.
10. Similarly: $(\forall t, t' : t \in \underline{stop}(N_B^W) \wedge t' \in \underline{stop}(N_C^W) : t^{N_0}\bullet = t'^{N_1}\bullet)$.
11. Since only transitions with non-τ labels in cp are connected to places in CA by the $\underline{cflat}$ function, $(\forall t : t \in T_B \wedge \ell_B(t) = \tau : ({}^{N_0}\bullet t \cap P_A = \emptyset) \wedge (t^{N_0}\bullet \cap P_A = \emptyset))$.

12. 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, $(\forall\, t : t \in T_C \wedge \ell_1(t) \notin \alpha(N_B^W) : ({}^{N_1}\bullet t \cap P_A = \emptyset) \wedge (t^{N_1}\bullet \cap P_A = \emptyset))$.
13. Similarly, one can show that $(\forall\, t, t' : t \in T_B \wedge t' \in T_C \wedge \ell_B(t) = \ell_C(t') : ({}^{N_0}\bullet t \cap P_A = {}^{N_1}\bullet t' \cap P_A) \wedge (t^{N_0}\bullet \cap P_A = t'^{N_1}\bullet \cap P_A))$.
14. $(\forall\, t, t' : t \in T_B \wedge t' \in \underline{start}(N_B^W)$: all non-trivial directed paths in N_0 from t to t' contain at least one occurrence of a transition in $\underline{stop}(N_B^W)$). This follows directly from the consistency of CA which is invariant under the replacements.
15. $N_C^W = \underline{cflat}(CA_c) \leq_{pj} CS_c$, because c is consistent. $CS_c \leq_{pj} cp = N_B^W$, because $(C, \underline{cmap})$ 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: $\underline{cflat}(CA'') \leq_{pj} CS_{c^t}$. □

Theorem 6 shows that a consistent system architecture is sound (i.e., no deadlocks, livelocks, and 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!

Consider for example 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 4. 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 Related work and future extensions

The results presented in this paper build upon earlier results on WF-nets [1–3], inheritance [4, 5, 8], and component-based software architectures [13]. Architecture definition languages such as ARMANI, Rapide, Darwin, Wright, and Aesop typically view software architectures statically [17], 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 [9] CORBA IDL's are extended with Petri nets to incorporate dynamic behavior. In the last decade several researchers [6, 14–16] 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 [8]. 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, 8]) and *transfer rules* to *migrate* instances from a superclass to a subclass and vice versa [5]. These features are very relevant for a both constructive and robust approach towards truly component-based software development.

In the future, we plan to extend our framework with other notions of inheritance. The three other notions of inheritance presented in [4, 8] can also be used to obtain complementary compositionality results. For example, the notion of *protocol inheritance* [4, 8], 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 [9] with our notion of inheritance. Finally, we plan to adapt our tools Woflan [23] and ExSpect [7] to serve the framework presented in this paper. Woflan can be used to check the requirements involving soundness and consistency. ExSpect can be used as a prototyping environment for experimenting with component-based software architectures.

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 Woflan, a verification tool which can be used to analyze many of the properties defined in this paper.

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. Finding Errors in the Design of a Workflow Process: A Petri-net-based Approach. In *Business Process Management: Models, Techniques, and Empirical Studies*, Springer-Verlag, Berlin, 1999 (to appear).
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. Computing Science Reports 99/06, Eindhoven University of Technology, Eindhoven, 1999.
6. 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.
7. Deloitte & Touche Bakkenist. ExSpect Home Page. http://www.exspect.com.
8. T. Basten. *In Terms of Nets: Systems Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998.

9. 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.
10. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
11. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
12. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
13. 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.
14. 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.
15. 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.
16. B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
17. 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.
18. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
19. 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.
20. B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/otl/technical/umlrt.html.
21. M. Shaw, G. David, and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
22. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
23. E. Verbeek and W.M.P. van der Aalst. Woflan Home Page. http://www.win.tue.nl/~woflan.