Niels Lohmann, Karsten Wolf (Hrsg.)

Algorithmen und Werkzeuge für Petrinetze

15ter Workshop, AWPN 2008
Rostock, 26.-27. September 2008

Proceedings

Preprints CS-03-08

Herausgeber:

Niels Lohmann, Karsten Wolf
Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{`niels.lohmann`, `karsten.wolf`}`@informatik.uni-rostock.de`

ISSN 0944-5900 (Preprint CS-03-08 Universität Rostock)
ISSN 1613-0073 (CEUR Workshop Proceedings)

Online-Proceedings verfügbar unter `http://CEUR-WS.org/Vol-380/`

BibTeX-Eintrag für Online-Proceedings:

```
@proceedings{awpn2008,
  editor    = {Niels Lohmann and Karsten Wolf},
  title     = {Proceedings of the 15th German Workshop on
               Algorithms and Tools for Petri Nets, AWPN 2008,
               Rostock, Germany, September 26--27, 2008},
  booktitle = {Algorithmen und Werkzeuge f\"ur Petrinetze},
  publisher = {CEUR-WS.org},
  series    = {CEUR Workshop Proceedings},
  volume    = {380},
  year      = {2008},
  url       = {http://CEUR-WS.org/Vol-380/}
}
```

Vorwort

Seit 1994 bietet der Workshop *Algorithmen und Werkzeuge für Petrinetze* ein gemeinsames Forum für Entwickler und Anwender petrinetzbasierter Technologie. Außerdem bildet er dank des traditionell geringen finanziellen Aufwands für die Teilnahme und der deutschsprachigen Ausrichtung eine Möglichkeit für Nachwuchswissenschaftler, Erfahrungen bei einer wissenschaftlichen Veranstaltung zu sammeln.

Im Jahr 2008 findet der Workshop in seiner 15ten Ausgabe erstmals in Rostock statt. Veranstalter ist wie immer die Fachgruppe *Petrinetze und verwandte Systemmodelle* der Gesellschaft für Informatik.

Es gab 16 eingereichte Beiträge, die alle nach kurzer Prüfung durch die Organisatoren in das Programm aufgenommen wurden. Ein Begutachtungsprozess fand dagegen, wie auch in den vergangenen Jahren, nicht statt. Wir hoffen, dass die Vorträge eine gelungene Grundlage für rege Diskussionen bieten.

Die Organisatoren danken der Fakultät für Informatik und Elektrotechnik der Universität Rostock für die finanzielle Untersützung der Ausrichtung.

August 2008

Niels Lohmann
Karsten Wolf

Steering Committee

Jörg Desel (Stellvertreter)	Katholische Universität Eichstätt-Ingolstadt
Ekkart Kindler	Technical University of Denmark
Kurt Lautenbach	Universität Koblenz-Landau
Robert Lorenz	Universität Augsburg
Daniel Moldt	Universität Hamburg
Rüdiger Valk	Universität Hamburg
Karsten Wolf (Sprecher)	Universität Rostock

Bisherige AWPN-Workshops

1. Berlin 1994	6. Frankfurt 1999	11. Paderborn 2004
2. Oldenburg 1995	7. Koblenz 2000	12. Berlin 2005
3. Karlsruhe 1996	8. Eichstätt 2001	13. Hamburg 2006
4. Berlin 1997	9. Potsdam 2002	14. Koblenz 2007
5. Dortmund 1998	10. Eichstätt 2003	15. Rostock 2008

Inhaltsverzeichnis

Oclets – scenario-based modeling with Petri nets

Dirk Fahland[⋆]

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany,
fahland@informatik.hu-berlin.de

Abstract. Scenario-based specifications are used for modeling highly-complex, distributed systems in terms of partial runs (*scenarios*) the system shall have. But it is difficult to derive an implementing, operational model from a given set of scenarios, especially if concepts like *anti-scenarios* which must not occur are used. In this paper, we present a novel model for scenario-based specifications with Petri nets including anti-scenarios; we provide an operational semantics for our model.

1 Operational semantics for scenario-based specifications

The paradigm of scenarios is widely accepted in protocol specifications using *message-sequence charts* (MSCs); *behavior* of highly-complex distributed systems is decomposed into reasonably sized artifacts called scenarios. Some classes of MSC specifications can be transformed into Petri nets [7], but usually an implementation has to be checked against an MSC specification. *Life-sequence charts* (LSCs) [5] extend the MSC paradigm by adding behavioral preconditions, anti-scenarios, and annotations to scenarios and single actions for enforcing their occurrence in the system. LSCs have a trace-based semantics (a set of charts accepts or rejects an execution trace) but, to our knowledge, there exists no complete operational semantics for the entire LSC language. Like for MSCs, subclasses of LSCs can be transformed into automata [4].

In this paper, we present an extension of Petri nets that with the key concepts of LSCs. Our model has operational semantics: For every set of scenarios, we can compute the branching process that implements the specification, extending the formal approach of [1]. Due to the very nature of Petri nets, we also introduce the notion of a local resource to LSC-style scenario-based specifications. Compared to other approaches for scenario-based specifications with Petri nets [6], we contribute the *anti-scenario* which explicitly forbids certain behavior in the system. In [3], we explained how our approach can be used for modeling adaptive processes in disaster management.

We will first sketch the key concepts of our approach in Sect. 2. We then explain our ideas related to a formal semantics for our model in Sect. 3 which we close with an outlook on future work. We assume the reader to be familiar with Petri nets and their branching time semantics in terms of branching processes; Esparza et al give a good introduction to these concepts in [2].

[⋆] The author's work is funded by the DFG-Graduiertenkolleg 1324 "METRIK".

2 Oclets - scenario-based specifications with Petri nets

A *scenario* specifies a possible course of (future) actions and the therein involved resources in the context of a larger system. Whether a scenario suits a given situation can be subject to further conditions. In our case, we conceive and formalize a scenario as a partial, partially ordered run (a labeled causal net) with a *behavioral* precondition. We define a *system model* as a set of scenarios describing sequentially connected, concurrent, mutually exclusive, and overlapping behavior. The *system behavior* shall be computed by composing its scenarios.

We formalize scenarios in our Petri net class of *oclets*. Let $Names = Actions \uplus Resources$ denote a set of labels.

Definition 1 (Oclet). *An oclet $o = \langle P, T, F, \ell, pre, type\rangle$ is a labeled, safe, elementary causal net $\langle P, T, F, \ell\rangle$ that labels places with resources and transitions with actions; o has a non-empty,* precondition *$pre \subseteq (P \cup T)$, that is causally closed ($\forall x \in pre :: {}^\bullet x \subseteq pre$), and a $type \in \{normal, anti\}$. The set $(P \cup T) \setminus pre$ is the* contribution *of o.*

A normal oclet describes a partial run that may occur in the system. An anti-oclet describes a partial run that may not be completed in the system; therefore an anti-oclet contributes exactly one place or transition (that must not occur). Figure 1 shows some (technical) example oclets. The system $\{o_1, \ldots, o_5\}$ shall yield the behavior that is formalized in the occurrence net β_5. The behavior of a set of oclets is constructed by repeatedly composing the oclets with a labeled occurrence net. An 'initial' occurrence net β_0 represents the initial state; composing β_i with an oclet o yields an occurrence net β_{i+1}.

Roughly, a normal oclet o is *composed* with a labeled occurrence net β, $\beta \oplus o$ by building the union of the nets, and merging two transitions (places) if they are labeled equally and have equally labeled predecessors. This is only allowed if o's precondition is found in β; all nodes of o's precondition are merged with nodes of β. To *compose* an anti-oclet o with β, $\beta \ominus o$, first compose o like for

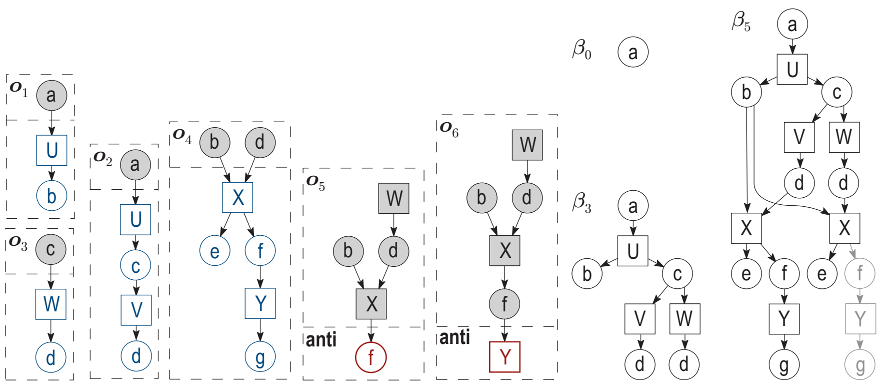


Fig. 1. Some example oclets $o_1, \ldots, o_6$ and three labeled occurrence nets $\beta_0, \beta_3, \beta_5$.

normal oclets, then remove the contribution of o and all successor nodes. Anti-oclets have priority: a node that was removed by an anti-oclet o^- is not added again by some other oclet o^+ as it is immediately removed by o^- again.

Consider the example of Fig. 1 with the initial occurrence net β_0 being a single place labeled a. Composing β_0 with oclet o_1, yields $\beta_1 := \beta_0 \oplus o_1$ which is isomorphic to o_1. $\beta_2 := \beta_1 \oplus o_2$ adds the post-place c to transition U and transition V with post-place d. In $\beta_3 := \beta_2 \oplus o_3$, transition W is added in conflict to V; see Fig. 1.

To compute $\beta_4 := \beta_3 \oplus o_4$, o_4 has to be added twice because there are two (conflicting) places d. Composing β_4 with anti-oclet o_5 removes f and successors Y and g from the branch that depends on W; the resulting occurrence net $\beta_5 := \beta_4 \ominus o_5$ is depicted in Fig. 1. Alternatively, composing with anti-oclet o_6 removes Y and successor g, but leaves f. The runs of β_5 are the runs of $\{o_1, \ldots, o_5\}$, the runs of β_6 are the runs of $\{o_1, \ldots, o_4, o_6\}$.

This informally sketched approach for scenario-based system specifications succeeds only if we can prove its formal consistency and show that branching processes (or rather a certain kind of labeled occurrence nets) are closed under our composition operations $\oplus$ and $\ominus$.

3 Formalizing oclets with canonically named nodes

Our oclet composition requires to ask frequently which nodes of an oclet o and an occurrence net β describe identical actions or resources, and, hence, must be merged. Formalizing this identity, and operations on labeled nets becomes tedious because two isomorphic nets may have disjoint, or overlapping sets of nodes. Identity can only be defined by relating labels of nodes to labels of neighboring nodes; this leads to graph isomorphism problems. Esparza and Heljanko use a formalization called *canonically named nodes* for formalizing branching processes of (safe) Petri nets [1]. In this section, we briefly sketch their key ideas and explain how we extend canonically named nodes for our model.

Canonically named nodes determine their identity by their labels and their predecessor: two nodes are identical if and only if they have identical labels and identical predecessors. The following formalization captures this *canonical identity*: The set $\mathcal{C}$ of *canonically named nodes* ($\mathcal{C}$-nodes) is defined inductively as the least set that contains $\langle a, \emptyset \rangle$ for every $a \in \mathit{Names}$ and if $x_1, \ldots, x_n \in \mathcal{C}$ and $a \in \mathit{Names}$ then $\langle a, \{x_1, \ldots, x_n\} \rangle \in \mathcal{C}$.

$\mathcal{C}$-nodes can be used as the base set of transitions and places of labeled Petri nets. A node $\langle \mathit{act}, X \rangle \in \mathcal{C}, \mathit{act} \in \mathit{Actions}$ is a *$\mathcal{C}$-transition* with label act, a node $\langle \mathit{res}, X \rangle \in \mathcal{C}, \mathit{res} \in \mathit{Resources}$ is a *$\mathcal{C}$-place* with label res. We use $\mathcal{C}$-nodes to formalize a specific class of labeled Petri nets.

Definition 2 ($\mathcal{C}$-net). *A labeled Petri net $N^{\mathcal{C}} = \langle P, T, F, \ell \rangle$ is a $\mathcal{C}$-*net *iff $P \subseteq \mathcal{C}$ are $\mathcal{C}$-places, $T \subseteq \mathcal{C}$ are $\mathcal{C}$-transitions, and for each $x := \langle a, X \rangle \in P \cup T$ holds:*

1. *if x is a $\mathcal{C}$-place, then X is a set of $\mathcal{C}$-transitions,*
2. *if x is a $\mathcal{C}$-transition, then X is a set of $\mathcal{C}$-places,*

3. $X \subseteq P \cup T$ *is the* preset *of* x*:* $y \in X$ *iff* $(y, x) \in F$*, and*
4. a *is the label of* x*:* $\ell(\langle a, X\rangle) = a$*.*

In a $\mathcal{C}$-net exist no two distinct, equally labeled nodes $\langle a, X\rangle, \langle a, Y\rangle$ with the same preset $X = Y$, this establishes the canonical identity of $\mathcal{C}$-nodes which we described above. This is a trivial mathematical consequence, but it has an interesting interpretation in branching processes: any two different actions (transitions) or resources (places) either have a different name, or a different causale. Esparza and Heljanko have shown that for this reason, $\mathcal{C}$-net structures are a good candidate to formalize branching processes (BP) of (safe) net systems [1], where the nodes of a net are labels to the nodes of the branching process.

Our oclet approach has a similar aim: construct branching-time artifacts that describe the behavior of a system. The difference is that we do not construct our artifacts from a net structure, but from oclets. Our construction does not only extend a branching process by adding a single transition (and its post-places) whenever the transition is enabled as in classical branching processes. The precondition of an oclet can be arbitrarily complex, and added nodes may have to be merged with the net. This means our formalization has to consider the causal structure of a labeled occurrence net and of an oclet *together*. To this end, we extend the $\mathcal{C}$-node approach of [1] as follows.

Operations on $\mathcal{C}$-nets and sets of $\mathcal{C}$-nodes The structure of a $\mathcal{C}$-net $N^{\mathcal{C}} = \langle P, T, F, \ell\rangle$ is completely encoded in its nodes, the information in its arcs F is redundant. Thus, the nodes $X_N^{\mathcal{C}} =_{\mathrm{df}} P \cup T$ of a $\mathcal{C}$-net $N^{\mathcal{C}}$ are sufficient to reconstruct F and, hence, $N^{\mathcal{C}}$. Because any two isomorphic $\mathcal{C}$-nets are identical, each (normal) Petri net N has a unique, isomorphic $\mathcal{C}$-net $N^{\mathcal{C}}$ which is completely encoded in $X_N^{\mathcal{C}}$.

This greatly simplifies our composition operation: the union of two sets of $\mathcal{C}$-nodes 'merges' canonically identical nodes by their identity. If we consider the sets $X_{o_1}^{\mathcal{C}}$ and $X_{o_2}^{\mathcal{C}}$ of $\mathcal{C}$-nodes of o_1 and o_2 in Fig. 1, the composition $\beta_2^{\mathcal{C}} := (\beta_0^{\mathcal{C}} \oplus o_1^{\mathcal{C}}) \oplus o_2^{\mathcal{C}}$ can be rephrased as the union $X_{\beta_2}^{\mathcal{C}} = X_{\beta_0}^{\mathcal{C}} \cup X_{o_1}^{\mathcal{C}} \cup X_{o_2}^{\mathcal{C}}$. For instance, $\langle \mathsf{a}, \emptyset\rangle$ and $\langle \mathsf{U}, \{\langle \mathsf{a}, \emptyset\rangle\}\rangle$ occur both in $o_1^{\mathcal{C}}$ and $o_2^{\mathcal{C}}$. But this approach does not work for o_3; $o_3^{\mathcal{C}}$ contains $\langle \mathsf{c}, \emptyset\rangle$, while $\beta_2^{\mathcal{C}}$ contains $\langle \mathsf{c}, \{\langle \mathsf{U}, \{\langle \mathsf{a}, \emptyset\rangle\}\rangle\}\rangle$.

Our proposed solution is to introduce variables into nodes with empty preset, e.g. $\langle \mathsf{c}, v\rangle$ such that the minimal nodes of an oclet which constitute the begin of a scenario can be assigned to other 'compatible' nodes 'further down' the occurrence net during the composition.

Let *Var* denote an (infinite) set of variables. The set $\mathcal{A}$ of *canonically named abstract nodes* ($\mathcal{A}$-nodes) differs to $\mathcal{C}$ in its induction base: For every $a \in \mathit{Names}$ and every $v \in \mathit{Var}$, $\langle a, v\rangle$ is an $\mathcal{A}$-node, and if $x_1, \ldots, x_n \in \mathcal{A}$ and $a \in \mathit{Names}$ then $\langle a, \{x_1, \ldots, x_n\}\rangle \in \mathcal{A}$. Correspondingly, the class of $\mathcal{A}$-nets can be defined; the variable takes the role of the empty pre-set, that is, a node $\langle a, v\rangle$ of an $\mathcal{A}$-net $N^{\mathcal{A}}$ has no predecessor in $N^{\mathcal{A}}$. Wlog. for all $\langle a_1, v_1\rangle, \langle a_2, v_2\rangle \in X_N^{\mathcal{A}}$ holds that $v_1 = v_2$ implies $a_1 = a_2$.

With this convention in mind, we transfer the pre-set notation ${}^{\bullet}(.)$ from $\mathcal{C}$-nodes (or Petri nets) to $\mathcal{A}$-nodes; we set ${}^{\bullet}\langle a, v\rangle =_{\mathrm{df}} v$. This canonically lifts all other notions like causal relation $\leq$, conflict $\sharp$, and concurrency $||$ from Petri nets and $\mathcal{C}$-nodes to $\mathcal{A}$-nodes. As a consequence, any two distinct nodes $\langle a_1, v_1\rangle, \langle a_2, v_2\rangle, v_1, v_2 \in \mathit{Var}$ are concurrent.

We introduce variables as place-holders for the pre-set of a $\mathcal{C}$-node. Thus an *assignment* α maps each variable v to a (possibly empty) set $\alpha(v)$ of $\mathcal{C}$-nodes, $\alpha : \mathit{Var} \to 2^{\mathcal{C}}$. If $x^{\mathcal{A}} \in \mathcal{A}$, then $x^{\mathcal{A}}[\alpha]$ denotes the $\mathcal{C}$-node that is obtained from $x^{\mathcal{A}}$ by simultaneously replacing every occurrence of each variable $v \in \mathit{Var}$ with $\alpha(v)$. This notion canonically lifts to sets $X^{\mathcal{A}} \subseteq \mathcal{A}$.

There is an important technical detail, that we have to consider: Let N be a safe, causal, labeled, elementary Petri net and let $X_N^{\mathcal{A}}$ be the corresponding set of $\mathcal{A}$-nodes of N. An assignment α is *feasible* on $X_N^{\mathcal{A}}$ iff for any two distinct minimal nodes $\langle a_1, v_1\rangle, \langle a_2, v_2\rangle \in X_N^{\mathcal{A}}$ holds: $\langle a_1, \alpha(v_1)\rangle \,||\, \langle a_2, \alpha(v_2)\rangle$. A feasible assignment guarantees that two concurrent nodes (like b and d of o_4 in Fig. 1) remain concurrent under the assignment.

We may now formalize our oclet composition operations.

Definition 3 (Enabling assignment). *Let $\beta^{\mathcal{C}}$ be a labeled $\mathcal{C}$-occurrence net, let $o^{\mathcal{A}}$ be an $\mathcal{A}$-oclet with precondition $pre^{\mathcal{A}}$. An α is* enabling for $o^{\mathcal{A}}$ in $\beta^{\mathcal{C}}$ *iff α is feasible and $pre^{\mathcal{A}}[\alpha] \subseteq X_\beta^{\mathcal{C}}$. Let $\mathit{enabled}(o^{\mathcal{A}}, \beta^{\mathcal{C}})$ denote the set of all enabling assignments for $o^{\mathcal{A}}$ in $\beta^{\mathcal{C}}$.*

Wlog. the set $\mathit{enabled}(o^{\mathcal{A}}, \beta^{\mathcal{C}})$ contains no two assignments α that differ only on variables which do not occur in $o^{\mathcal{A}}$.

As an example consider o_1 and β_0 of Fig. 1: $pre_{o_1}^{\mathcal{A}} = \{\langle \mathsf{a}, v_1\rangle\}$, $X_{\beta_0}^{\mathcal{C}} = \{\langle \mathsf{a}, \emptyset\rangle\}$. The assignment that maps v_1 to $\emptyset$ is enabling for o_1 in β_0. Oclet o_4 has two qualitatively different enabling assignments in β_3.

A further notion which we need for the composition is the *causal past* $\lfloor x \rfloor$ of a $\mathcal{C}$-node x with $\lfloor x \rfloor =_{\mathrm{df}} \{y \in \mathcal{C} \mid y \leq x\}$; this notion also lifts to sets of $\mathcal{C}$-nodes.

Definition 4 (Oclet composition). *Let $\beta^{\mathcal{C}}$ be a labeled $\mathcal{C}$-occurrence net. Let $o^{\mathcal{A}}$ be an $\mathcal{A}$-oclet with $\mathit{enabled}(o^{\mathcal{A}}, \beta^{\mathcal{C}}) = \{\alpha_1, \ldots, \alpha_k\}$.*

If $o^{\mathcal{A}}$ is a normal oclet, then the composition of $\beta^{\mathcal{C}}$ with $o^{\mathcal{A}}$ *yields the $\mathcal{C}$-net $\beta_2^{\mathcal{C}} =_{\mathrm{df}} \beta^{\mathcal{C}} \oplus o^{\mathcal{A}}$ with $X_{\beta_2}^{\mathcal{C}} =_{\mathrm{df}} X_\beta^{\mathcal{C}} \cup (X_o^{\mathcal{A}}[\alpha_1] \cup \ldots \cup X_o^{\mathcal{A}}[\alpha_k])$.*

If $o^{\mathcal{A}}$ is an anti-oclet with contribution $\{y_o\} = X_o^{\mathcal{A}} \setminus pre_o^{\mathcal{A}}$, then the composition of $\beta^{\mathcal{C}}$ with $o^{\mathcal{A}}$ *yields the $\mathcal{C}$-net $\beta_2^{\mathcal{C}} =_{\mathrm{df}} \beta^{\mathcal{C}} \ominus o^{\mathcal{A}}$ with $X_{\beta_2}^{\mathcal{C}} =_{\mathrm{df}} \{x \in X_\beta^{\mathcal{C}} \mid \lfloor x \rfloor \cap (y_o[\alpha_1] \cup \ldots \cup y_o[\alpha_k]) = \emptyset\}$.*

Consider $o_1^{\mathcal{A}}$ and $\beta_0^{\mathcal{C}}$ of our example; $X_{o_1}^{\mathcal{A}} = \{p_1^{\mathcal{A}}, t_1^{\mathcal{A}}, p_2^{\mathcal{A}}\}$ with $p_1^{\mathcal{A}} = \langle \mathsf{a}, v_1\rangle$, $t_1^{\mathcal{A}} = \langle \mathsf{U}, \{p_1^{\mathcal{A}}\}\rangle$, $p_2^{\mathcal{A}} = \langle \mathsf{b}, \{t_1^{\mathcal{A}}\}\rangle$. The enabling assignment $\{\alpha\} = \mathit{enabled}(o_1^{\mathcal{A}}, \beta_0^{\mathcal{C}})$ yields $X_{o_1}^{\mathcal{A}}[\alpha] = \{p_1^{\mathcal{C}}, t_1^{\mathcal{C}}, p_2^{\mathcal{C}}\}$ with $p_1^{\mathcal{C}} = \langle \mathsf{a}, \emptyset\rangle$ etc. Thus the composition $\beta_0^{\mathcal{C}} \oplus o_1^{\mathcal{A}}$ yields exactly $X_{o_1}^{\mathcal{A}}[\alpha]$, merging the two places labeled a.

The composition with an anti-oclet is formally more involved, but straight forward: All nodes of $\beta_5^{\mathcal{C}}$ in Fig. 1 including the greyly shaded ones constitute $\beta_4^{\mathcal{C}}$ where $o_5^{\mathcal{A}}$ has one enabling assignment $\{\alpha\} = \mathit{enabled}(o_5^{\mathcal{A}}, \beta_4^{\mathcal{C}})$ mapping the variables of $\langle \mathsf{b}, v_1\rangle$ and $\langle \mathsf{W}, v_2\rangle$ of $o_5^{\mathcal{A}}$ to $\{\langle \mathsf{U}, \{p_{\mathsf{a}}^{\mathcal{C}}\}\rangle\}$ and $\{\langle \mathsf{c}, \{t_{\mathsf{U}}^{\mathcal{C}}\}\rangle\}$ of $\beta_4^{\mathcal{C}}$, respectively. The contributed node of $o_5^{\mathcal{A}}$ is $y_{o_5} = \langle \mathsf{f}, \{t_{\mathsf{X}}^{\mathcal{A}}\}\rangle$; α maps y_{o_5} to the right-most node $\langle \mathsf{f}, \{t^{\mathcal{C}}_{\mathsf{X},2}\}\rangle = y_{o_5}\,[\alpha]$ of $\beta^{\mathcal{C}}_4$. All nodes of $\beta^{\mathcal{C}}_4$ which have this node in their causal past are to be removed, i.e. $\langle \mathsf{f}, \{t^{\mathcal{C}}_{\mathsf{X},2}\}\rangle$ itself and all nodes reachable from it via the flow-relation. This results in $\beta^{\mathcal{C}}_5$.

With this formalization one can show that labeled $\mathcal{C}$-occurrence nets are closed under composition with $\oplus$ and $\ominus$. From the set theoretic definitions of $\ominus$ follows that $(\beta \ominus o_1) \ominus o_2 = (\beta \ominus o_2) \ominus o_1$ for any $\mathcal{C}$-occurrence net β and any two $\mathcal{A}$-anti-oclets o_1 and o_2. $(\beta \oplus o_1) \oplus o_2 = (\beta \oplus o_2) \oplus o_1$ for normal oclets o_1, o_2 holds only if o_2 does not introduce new enabling assignments for o_1. The behavior of a set of oclets is defined as its *$\mathcal{C}$-unfolding*:

Definition 5 ($\mathcal{C}$-unfolding). *Let O be a set of oclets with $\{o_1, \ldots, o_k\}$ and $\{o_{k+1}, \ldots, o_l\}$ being the normal oclets and the anti-oclets of O, respectively. Let β_0 be a $\mathcal{C}$-occurrence net. The fixed point of the sequence $\langle \beta_0, \beta_1, \beta_2, \ldots\rangle$ with $\beta_{i+1} =_{\mathrm{df}} (\beta_i \oplus o_1 \oplus \ldots \oplus o_k) \ominus o_{k+1} \ominus \ldots \ominus o_l$ is the* $\mathcal{C}$-unfolding *of O.*

Summary and Future Work. Definition 5 concludes the presentation of our basic model for scenario-based specifications with Petri nets. The presented expressive means allow specifying complex behavior in terms of partial runs which may or must not occur.

The basic model has already been implemented in the *Graphical Runtime EnvironmenT for Adaptive systems (GRETA)*. Next, we will introduce further LSC features like *hot* and *cold* annotations for specifying which actions must occur and which states are legal final states. Further, we plan to introduce the notion of an *interface* to specify system composition, and system interaction. Finally, the question which Petri net has the same behavior as a given set of oclets shall be addressed.

References

1. J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking.* Springer-Verlag, 2008.
2. J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. In *TACAS 1996*, volume 1055 of *LNCS*, pages 87–106. Springer-Verlag, 1996.
3. D. Fahland and H. Woith. Towards Process Models for Disaster Response. In *Proceedings of PM4HDPS 2008, co-located with BPM'08*, Milan, Italy, September 2008. Accepted.
4. D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *CIAA 2000*, volume 2088 of *LNCS*, pages 1–33. Springer-Verlag, 2001.
5. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine.* Springer-Verlag New York, Inc., 2003.
6. R. Lorenz, R. Bergenthum, J. Desel, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. In *ACSD 2007*, pages 157–166. IEEE Computer Society, 2007.
7. M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Netcharts: Bridging the gap between HMSCs and executable specifications. In *CONCUR 2003*, volume 2761 of *LNCS*, pages 296–310. Springer-Verlag, 2003.

EWFN - A Petri Net Dialect for Tuplespace-based Workflow Enactment[*]

Daniel Martin, Daniel Wutke, and Frank Leymann

Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstrasse 38, 70569 Stuttgart, Germany
{martin,wutke,leymann}@iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de

Abstract Petri nets are a formalism for describing systems where interactions between active components – so-called transitions – are modeled as exchanges of tokens over passive places. Whether a transition may fire is solely dependent on the availability of tokens in its incoming places; similarly a transition forwards control to subsequent transitions by storing tokens in their respective input places. This interaction model of strong decoupling through local actions and local effects makes distributed systems modeled via Petri nets highly extensible. In this paper, we present the syntax and semantics of a model that leverages the extensibility provided by Petri nets for representing BPEL processes in a way that enables their distributed and decentralized execution using tuplespace middleware. Said middleware implements the proposed Petri net dialect and therefore allows for direct, distributed execution of the modeled processes.

Key words: Petri nets, Tuplespaces, Workflow

1 Introduction

Petri nets were originally designed as a model for arbitrary extensible computer architectures i.e. machines that consist of many individual modules, each of them responsible for a particular task of the overall system. Adding a new module has no impact on the existing ones, their performance characteristics for instance do not change at all. Three underlying design principles [1] facilitate this behavior: (i) there is no central point of control, especially, there is no central clock. Moreover, synchronizing clocks between modules is considered bad design and should be avoided in any case. (ii) Each action is triggered locally, and has only local effect; i.e. enabling of a transition only depends on its input places, firing of a transition only effects its output places. There is no way to access the global state of the system. (iii) Petri nets are inherently asynchronous in nature, communication solely happens over local interfaces in a peer to peer like manner.

[*] This work is supported by the EU funded project *TripCom* (FP6-027324)

These principles build the foundation for our model, that is naturally based on Petri nets. In their spirit, we define a set of individual components and the communication between them. The communication middleware that facilitates component interaction during execution of the model is based on tuplespaces, since (i) they closely resemble the design properties of petri nets in terms of loose coupling and asynchronous communication [2] and (ii) each element of a Petri net can be directly mapped to an entity in a tuplespace based system (either a component, a tuple or a tuplespace).

Tuplespace technology has its origin in the *Linda coordination language*, defined in [3] as a parallel programming extension for programming languages for the purpose of separating coordination logic from program logic, i.e. the actual application code. The Linda concept is built on the notion of a *tuplespace*, a piece of memory that is shared among all interacting parties. A user interacts with the tuplespace by storing and retrieving *tuples* (i.e. an ordered list of typed fields) via a simple interface: tuples can be (i) stored (using the *write* operation), (ii) retrieved destructively (*take*) and (iii) non-destructively (*read*). Tuples are retrieved using a template mechanism, e.g. by providing values of a subset of the typed fields of the tuple to be read, similar to *query by example* [4] ("associative addressing"). Using tuplespace-based coordination, execution of a component's computational logic is triggered when tuples matching the templates registered by the respective component are present in the tuple space. Thus, the templates a component uses to consume tuples and the tuples it produces represent its coordination logic.

In this paper, we define a variant of Petri nets, called *Executable Workflow Networks (EWFN)*, specifically designed to represent BPEL workflows and being *executed* "natively" on an extended, Linda-like tuplespace system. The basis for our model are colored, non-hierarchical Petri nets (CPN) [5] and Boolean networks [6]. We present an extension of the model presented in [7], building upon the syntax and concentrate on the description of the semantics.

2 Syntax

Definition 1 (EWFN). *An EWFN is a directed, bipartite graph*

$$EWFN = (\Sigma, P, T, F, X, A, M_0, L_w)$$

$\Sigma = \{CF, DATA \times \mathbb{N}, DATA \times \mathbb{N} \times \text{String}, \ldots, \epsilon\}$ denotes the set of tokens (tuples). Note that Σ comprises two different categories of tokens: (i) control flow tokens $CF = (\text{“CF”} \times S \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$ with $S = \{\text{“POS”}, \text{“NEG”}, \text{“FAIL”}\}$ denoting either "positive", negative (a.k.a *dead path*, a special form of "negative" control flow necessary for dead path elimination in WS-BPEL) or control flow initiated by a failure, and (ii) data tokens representing BPEL variables and process meta-data. The three integer fields represent *processID*, *instanceID* and *scopeID* in order to be able to distinguish between process models, process instances and scopes that were initiated by event-handlers. Data tokens consist of the generic data tuple (denoted as $DATA = (\text{“DATA”} \times \mathbb{N} \times \mathbb{N})$) concatenated with *variable* definitions (in tuple form) from the respective process. We represent arbitrary structured data by serializing its tree-based representation (e.g. in the form of an XML-DOM [8]) into nested tuples. Furthermore, Σ contains the "empty" tuple ϵ used to denote that actually no tuple is produced.

Note that like most other formalizations of Petri nets, our description is based on multi-sets, we therefore define the operators $+$, $-$, etc. to be defined on multi-sets as well.

P is a finite set of *places* and T a finite set of *transitions* such that $P \cap T = \emptyset$.

$F \subseteq (T \times P) \cup (P \times T \times R)$, with $R = \{\mathit{read}, \mathit{take}\}$ is a set of arcs known as *flow relation*. The set F is subject to the constraint that no arc may connect two places or two transitions. The arc types correspond to classical Linda operations [3]: *write* (a.k.a *out*) arcs go from transitions to places (i.e. are member of the set $(T \times P)$), whereas *read* (a.k.a *rd*) and *take* (a.k.a *in*) arcs go from places to transitions, with arc inscription R denoting the type of arc. *Take* arcs are known from classical Petri nets (i.e. they destructively consume tokens from places). *Read* arcs (a.k.a test arcs) [9] in contrast allow a transition to non-destructively read a token from a place.

X is a set of *templates* in tuple form, that may either contain a wildcard ($\star$) or a concrete value as element.

$A : (P \times T \times R) \rightarrow X$ is a function that assigns templates to incoming arcs of a transition such that $\forall (p, t, r) \in F \cap (P \times T \times R) : A((p, t, r)) \in X$. Sometimes, we use A without the last parameter, as a shortcut to access the template assigned to an arc pointing to a transition. In these cases, it is not important whether the template is used in a *read* or a *take* operation.

$M_0 : P \rightarrow \Sigma_{MS}$ is an initialization function that assigns a multi-set over Σ to places such that $\forall p \in P : M_0(p) \in \Sigma_{MS}$ This function initializes the network by assigning a multi-set of colored tokens to each place. It is also allowed that the expression is missing, i.e. a place is initialized with the empty color multi-set.

$L_w : (T \times P) \rightarrow \Sigma$ is the Linda write function that determines the token to be written by each outgoing arc of a transition. Writing an empty tuple (ϵ) means that no tuple is written at all.

Definition 2 (tuple element). *A tuple element TE is a tuple (p, tu), $p \in P$, $tu \in \Sigma$*

Definition 3 (marking). *A* marking $M \in TE_{MS}$ *is a multi-set (denoted as $_{MS}$) over tuple elements. Each place may contain one or more equal tuple elements, thus the marking is defined as a multi-set. Note that we may also use M as a function such that $\forall p \in P : M(p) \in \Sigma_{MS}$*

Definition 4 (L_r). *Linda read operations (destructive and non-destructive) are formalized as a function $L_r : X \times \Sigma_{MS} \rightarrow \Sigma$. According to Linda's semantics [3], only one tuple is returned regardless the number of matching tuples. It is not determined which tuple of the set of matching tuples is returned: $L_r(te, tu_{MS}) = tu \in tu_{MS} | tu \approx te$.*

$\approx$ is a binary relation over the sets Σ and X, specifying if a template matches a tuple: $\approx \subseteq \Sigma \times X$.

$$(tu, te) \in \approx \text{ iff } |tu| = |te| \wedge (\forall n \in 1..|te| : \pi_n(te) = \pi_n(tu) \vee \pi_n(te) = \star)$$

$\pi_i(t)$ returns a projection to the i^{th} component of a tuple tu, $|tu|$ denotes the *size* of a tuples, i.e. the number of elements it contains.

A template therefore is a tuple that has either a wildcard (denoted by the $\star$ character) or a concrete value on each position. A template matches a tuple iff (i) both have the same number of elements and (ii) each concrete value in the tuple equals the value on the same position in the template, or (iii) the template has a wildcard on this position.

Definition 5 (strongly connected). *An EWFN is called* strongly connected *[10] iff for every pair of nodes (places and transitions) x and y there is a firing sequence leading from x to y.*

Similar to WF-nets [10], an EWFN has two special kinds of transitions: t_a and t_o. There is no arc pointing to t_a, i.e. $\bullet t_a = \emptyset$, similarly, t_o has no outgoing arcs, i.e. $t_o \bullet = \emptyset$. If we add a place $p^\star$ to the EWFN to connect transition t_o with t_a (i.e. $\bullet p^\star = \{t_o\}$ and $t^\star \bullet = \{t_a\}$), then the resulting net is *strongly connected*. Transitions of type t_a do not have a precondition, i.e. are formally allowed to fire any time. We use such transitions to create process instances (i.e. create a CF tuple with new instance id) in our model. Similarly, t_o does not have outgoing transitions, this transition only consumes tokens from the EWFN and is used to log process instance termination.

3 Semantics

A transition $t \in T$ that executes a *destructive* read operation (a.k.a take) changes marking M_1 to M_2 as follows:

$$\forall p \in \bullet t : M_2(p) = M_1(p) - L_r(A((p, t, \text{“take”})), M_1(p))$$

A transition $t \in T$ that executes a *non-destructive* read operation in contrast, does not have any effect on the marking:

$$\forall p \in \bullet t : M_2(p) = M_1(p)$$

The set of places that have arcs pointing to transition t is denoted as $\bullet t = \{p|pFt\}$, the set of transitions that have arcs pointing to place p is denoted as $\bullet p = \{t|tFp\}$, with F being the flow relation. $t\bullet$ and $p\bullet$ are defined accordingly.

Definition 6 (enabled). *A transition $t \in T$ is called* enabled *in marking M iff*

$$\forall p \in \bullet t : L_r(A((p, t)), M(p)) \neq \emptyset$$

It is important to notice that the templates of read operations may overlap, i.e. if two different transitions destructively read from the same place with templates that (partially) match the same tuple, a conflict is created. According to Linda semantics [3], this conflict is resolved non-deterministically. Clearly, non-deterministic decisions are not suitable for workflow definitions. That is why we extend the enablement rule of a transition in EWFNs to be "conflict free" enabled. If there are transitions in an EWFN that cause conflicts, the EWFN is not valid.

Definition 7 (conflict-free enabled). *A transition $t \in T$ is called* conflict-free enabled *in marking M iff*

$$\begin{aligned}
&t \textit{ is enabled} \wedge \\
&\forall t' \in (\bullet t) \bullet \setminus \{t\} : t' \textit{ is not enabled} \vee \\
&\forall p \in \bullet t \cap \bullet t' : L_r(A((p,t)), M(p)) \neq L_r(A((p,t')), M(p)) \vee \\
&(\forall p \in \bullet t \cap \bullet t' : L_r(A((p,t)), M(p)) = L_r(A((p,t')), M(p)) \wedge \\
&\qquad\qquad\qquad\qquad (p, t, \textit{"read"}) \in F \wedge (p, t', \textit{"read"}) \in F)
\end{aligned}$$

Intuitively, a transition t is conflict free enabled if all other transitions t' that share an input place with this transition are not enabled, they do not read the same tuple or they read the same tuple but all issue non-destructive read operations only on the place in question. Since we describe executable workflows, conflict situations where the actual decision is not-determined and ultimately lead to "confusion" [1] are not desired in our model.

The property of conflict-freeness however is defined on enablement of a transition, i.e. it can only be checked during runtime. The following, alternative definition defines conflict-freeness of a transition based on the templates of the read operations it issues, thus allows to check for conflict-freeness of an EWFN on the syntactical level, i.e. check an EWFN after transformation from BPEL.

Definition 8 (conflict-free transition). *A transition $t \in T$ is called* conflict-free *iff*

$$\begin{aligned}
&\forall p \in \bullet t \ \forall t' \in p \bullet \setminus \{t\} : A((p,t)) \cap A((p,t')) = \emptyset \vee \\
&\qquad\qquad\qquad\qquad ((p, t, \textit{"read"}) \in F \wedge (p, t', \textit{"read"}) \in F)
\end{aligned}$$

A transition t is conflict-free iff the intersection of templates of the read operations from different transitions reading from shared places with t is empty, or every transition in question issues only non-destructive read operations. For the reasons mentioned before, we enforce all transitions in an EWFN to be conflict free.

Definition 9 (satisfied). *A template $te \in X$ is called* satisfied *on multi-set tu_{MS} iff $L_r(te, tu_{MS}) \neq \emptyset$. This can also be written as function $Sat : X \times \Sigma_{MS} \rightarrow \mathbb{B}$.*

$$Sat(te, tu_{MS}) = \begin{cases} \texttt{true}, \textit{ if } L_r(te, tu_{MS}) \neq \emptyset \\ \texttt{false}, \textit{ otherwise} \end{cases}$$

Definition 10 (fire). *A transition $t \in T$ that is enabled in marking M_1* may fire *and change marking M_1 to M_2 as follows:*

$$\forall p \in \bullet t \cup t\bullet : M_2(p) = M_1(p) - \sum_{p_n \in \bullet t} L_r(A((p_n, t)), M_1(p_n)) \;+ \sum_{p_n \in t\bullet} L_w(t, p_n)$$

Note that in this definition, the operators $+$, $-$ and $\sum$ are defined on multi-sets, removing and adding tuples from the multi-set respectively.

We extend the template matching from Definition 4 to be able to understand *join variables* as fields in a template tuple. Join variables allow to express a restriction on the enablement of a transition such that it is only enabled if every template of its read/take operations that use a *join variable* is satisfied and the tuple elements on the position of the join variable are equal for each join variable. Note that for the matching itself a join variable is treated as wildcard ($\star$).

Consider the join of two threads of control flow of the same workflow instance and process model, identified by the ids *iid* and *pid* respectively:

$$te_1 = (\text{“}CF\text{”}, ?\mathrm{pid}, ?\mathrm{iid})$$
$$te_2 = (\text{“}CF\text{”}, ?\mathrm{pid}, ?\mathrm{iid})$$

The transition using two separate take operations with te_1 and te_2 as templates is only enabled if there are tuples available in both incoming places that have equal values on their second and third position.

Definition 11 (join matching). *A transition $t \in T$ that uses* join variables *in its template operations is enabled in marking M iff*

$$\begin{aligned}
&\forall p \in \bullet t : L_r(A((p,t))[?*/\star], M(p)) \neq \emptyset \wedge \\
&\forall p_1, p_2 \in \bullet t \; \exists n \in \mathbb{N} : \pi_n(A((p_1,t))) \textit{ is join variable} \wedge \\
&\pi_n(A((p_2,t))) \textit{ is join variable} \wedge \\
&\pi_n(A((p_1,t))) = \pi_n(A((p_2,t))) \wedge \\
&\qquad\qquad \pi_n(L_r(A((p_1,t)), M(p_1))) = \pi_n(L_r(A((p_2,t)), M(p_2)))
\end{aligned}$$

The treatment of join variables for the actual matching is expressed as $[?*/\star]$, meaning that every variable that starts with a ? is replaced by a wildcard ($\star$).

For space reasons, we omit the usual definitions for firing sequence, reachability, liveness, boundedness, safeness well structuredness and well-formedness [10] for EWFNs.

4 Conclusion and Future Work

In this paper, we presented a tuplespace-based Petri net dialect that is natively executable on a tuplespace system, i.e. each element of the Petri net has an equivalent element or operation on a tuplespace. An EWFN therefore is a kind of “byte code” for tuplespace-based applications; they can be designed using EWFNs and then directly transformed to a running application.

The main idea behind the development of EWFNs however is their use in decentralized workflow enactment. We are working on a BPEL engine that transforms BPEL files to EWFNs and then executes them based on tuplespaces. Each tuplespace can reside on a different machine in the network, thus the engine and even the execution of a single process instance may be arbitrarily distributed. The key enabler for this architecture are Petri nets and their inherent properties such as: no central point of control, local actions, local effects, asynchronous interaction.

References

1. Reisig, W.: Petri nets: An Introduction. Springer-Verlag New York, Inc. New York, NY, USA (1985)
2. Aldred, L., van der Aalst, W., Dumas, M., ter Hofstede, A.: On the Notion of Coupling in Communication Middleware. Proc. of Intl. Symposium on Distributed Objects and Applications (DOA) (2005) 1015–1033
3. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
4. Zloff, M.: Query by Example. AFIPS Conference Proceedings, 1975 National Computer Conference **44**
5. Jensen, K.: Coloured Petri Nets, Vol. 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag (1992)
6. Langner, P., Schneider, C., Wehler, J.: Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen. Wirtschaftsinformatik **39**(5) (1997) 479–489
7. Wutke, D., Martin, D., Leymann, F.: Model and infrastructure for decentralized workflow enactment. Proceedings of the 23rd ACM Symposium on Applied Computing (SAC'08) (2008)
8. Le Hors, A., et al.: Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation (2004)
9. Vogler, W., Semenov, A., Yakovlev, A.: Unfolding and Finite Prefix for Nets with Read Arcs. Proceedings of the 9th International Conference on Concurrency Theory (1998) 501–516
10. van der Aalst, W.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66

Ein Petrinetz - Modell zur Informationsübertragung per Dialog

Markus Huber[1], Christian Kölbl[1], Robert Lorenz[2], and Günther Wirsching[1]

[1] Katholische Universität Eichstätt - Ingolstadt
{firstname.lastname}@ku-eichstaett.de
[2] Universität Augsburg
robert.lorenz@informatik.uni-augsburg.de

Zusammenfassung Wir stellen ein abstraktes Modell zur Informationsübertragung per Dialog vor. Dieses Modell erweitert das Grundmodell eines Kommunikationssystems von Shannon um die Möglichkeit der Nachfrage innerhalb einer Informationsübertragung. Für die Modellierung von Information und Informationsbestandteilen führen wir Merkmal-Werte-Relationen ein. Die Steuerung des Informationsflusses repräsentieren wir durch ein farbiges Petrinetz. Hierbei konzentrieren wir uns auf einen Mensch-Maschine-Dialog und die Modellierung der Funktionalität der Maschine. Eine zukünftige industrielle Anwendung liegt im Bereich von Dialogsystemen zur Sprachverarbeitung.

1 Einleitung

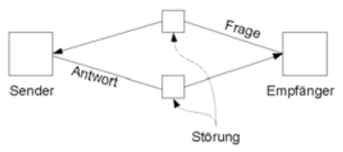


Abbildung 1. Dialogische Umsetzung des Shannonschen Kommunikationsmodell von 1948 (Fig. 1 in [3])

Mit der Erfindung des Telefons begann langsam das Bedürfnis zu wachsen, den Prozess der Übertragung von Nachrichten und Information genauer zu verstehen. Ein Meilenstein auf diesem Weg war Shannon's 1948 erschienene Arbeit "A Mathematical Theory of Communication" [3], in der insbesondere ein mathematisches Modell für den Umgang mit Störungen des Sender-Empfänger-Kanals vorgestellt wurde. In vielen realen Situationen ist der Kanal zwar Störungen unterworfen, funktioniert aber im Prinzip in beiden Richtungen so gut, dass eine wesentlich sicherere Informationsübertragung per Frage-Antwort-Dialog möglich ist (Abb. 1). Das Ziel dieser Arbeit ist einen Beitrag zur mathematischen Modellierung der dialogischen Vermittlung von Information zu leisten. Um die Zuverlässigkeit unseres mathematischen Modells zu gewährleisten, ist es unsere Absicht, das Dialogmodell schließlich so genau zu beschreiben, dass eine technische Realisierung möglich ist und zu Dialogsystemen führt, die den zur Zeit üblichen Sprachdialogsystemen, wie sie etwa bei Hotlines oder zur Durchführung telefonischer Überweisungen eingesetzt werden, weit überlegen sind.

Hierbei gehen wir *top-down* vor und behandeln in dieser Arbeit zwei grundlegende Aspekte:

1. Die Modellierung von *Information* und *Informationsbestandteilen* durch *Merkmal-Werte-Relationen*, und

2. die Steuerung des Informationsflusses durch ein farbiges Petrinetz.

Der erste Aspekt erlaubt es, bei der Repräsentation von Information mögliche Kanalstörungen und Mißverständnisse angemessen zu berücksichtigen, und geht damit über das verbreitete Modell *semantic slots* [1] hinaus. Die im zweiten Aspekt angesprochene Dialogsteuerung durch ein Petrinetz bildet einen vernünftigen formalen Rahmen, in den prinzipiell alle bekannten Dialogstrategien integriert werden können.

2 Systemmodellierung

In diesem Abschnitt wird ein abstraktes Modell zur Informationsübertragung per Dialog vorgestellt. Da es möglich ist, dass übertragene Information nicht genau verstanden wird, kann mit einer Nachfrage reagiert werden. Dabei wird eine Erwartung darüber generiert, was der Dialogpartner inhaltlich auf die Nachfrage antworten könnte. Zur Modellierung von Informationsbestandteilen führen wir sog. *Merkmal-Werte-Relationen (MWRen)* ein. Um einerseits die Erwartung an den Inhalt übertragener Information und andererseits die Sicherheit mit der Information verstanden wurde ausdrücken zu können, können MWRen gewichtet werden. Zur Modellierung des Informationsflusses im Dialog verwenden wir ein farbiges Petrinetz. Im Vergleich zu (bisher überwiegend verwendeten) Automaten hat das Petrinetz Vorteile in Mächtigkeit, Kompaktheit, Lesbarkeit, Wartbarkeit und Erweiterbarkeit.

Es handelt sich um einen bisher noch vollständig abstrakten Entwurf als ersten Schritt einer Top-Down-Modellierung. Wir interpretieren dabei die Transitionen als abstrakte Funktionen. Dazu definieren wir formal, was die Eingabe- und Ausgabeparameter dieser Funktionen sind (diese werden als farbige Marken in den Stellen modelliert), interpretieren die Ein- und Ausgabewerte der Funktionen und beschreiben deren grundsätzliche funktionale Abhängigkeit. Die Implementierung derselben, z.B. durch Transitionsverfeinerung, ist Gegenstand weiterer Forschungsarbeiten.

Im folgenden Abschnitt werden die grundlegenden Notationen eingeführt. Im Anschluss daran definieren und beschreiben wir den Begriff der Merkmal-Werte-Relation, der für die Systembeschreibung im dritten Abschnitt von zentraler Bedeutung sein wird.

2.1 Grundlegende Notationen

Wir beginnen mit einigen grundlegenden mathematischen Notationen. Mit $\mathbb{N}$ bezeichnen wir die Menge der *nicht-negativen ganzen Zahlen*, mit $\mathbb{R}$ die Menge der *reellen Zahlen* und mit $\mathbb{R}^+$ die Menge der nicht-negativen reellen Zahlen. Ein Wahrscheinlichkeitsmaß π auf einer endlichen Menge A ist eine Abbildung $\pi : A \to [0,1]$ mit $\sum_{a \in A} \pi(a) = 1$. Für eine endliche Menge A bezeichnet $A+$ wie üblich die Menge aller Worte über A. Eine *Multi-Menge* über einer Menge A ist eine Funktion $m : A \to \mathbb{N}$. Für ein $a \in A$ bezeichnet $m(a)$ die Anzahl von a's in m. Für eine binäre Relation $R \subseteq A \times A$ über einer Menge A ist R^+ der *transitive Abschluss* von R. Eine binäre Relation R über A heißt *linkstotal*, falls für jedes $a \in A$ ein $b \in A$ existiert mit $(a, b) \in R$. Ein *gerichteter Graph G* ist ein Paar $G = (A, \to)$, wobei A eine endliche Menge von *Knoten* und $\to \subseteq A \times A$ die Menge der *Kanten* ist. Wie üblich schreiben wir auch $a \to b$

für $(a, b) \in\rightarrow$. Für $a \in A$ bezeichnet ${}^{\bullet}a = \{a' \in A \mid a' \rightarrow a\}$ den Vorbereich und $a^{\bullet} = \{a' \in A \mid a \rightarrow a'\}$ den Nachbereich von a. Eine endliche Folge von Knoten $a_0 \dots a_n$ ($n \in \mathbb{N}$) mit $a_{i-1} \rightarrow a_i$ ist ein *Pfad* von a_0 nach a_n. Ein Pfad $a_0 \dots a_n$ mit $a_0 = a_n$ ist ein *Zyklus*. Eine *partielle Ordnung* ist ein gerichteter Graph $(A, <)$, wobei $<$ irreflexiv und transitiv ist. Eine Relation R über einer Menge A lässt sich als gerichteter Graph (A, R) auffassen und umgekehrt. Bekanntlich ist eine Relation R genau dann zyklenfrei, wenn R^+ irreflexiv, d.h. eine partielle Ordnung, ist.

Zur Modellierung der Dialogsteuerung verwenden wir farbige Petrinetze [2]. Aus Platzgründen führen wir farbige Petrinetze hier nur semi-formal ein. Ein *farbiges Petrinetz* besteht aus einer endlichen Menge von Stellen S, einer endlichen Menge von Transitionen T ($S \cap T = \emptyset$), einer Menge von Kanten $F \subseteq (S \times T) \cup (T \times S)$ und einer endlichen Menge von Farbmengen Σ. Eine Farbmenge $C \in \Sigma$ lässt sich als Wertebereich eines Datentyps auffassen. Einen Wert $c \in C$ bezeichnet man als Farbe. Jeder Stelle s ist eine Farbmenge $C(s)$ zugeordnet, welche den Datentyp der Marken festlegt, die in dieser Stelle liegen dürfen. Die *initiale Markierung* m_0 legt fest, wieviele Marken welcher Farbe am Anfang in einer Stelle s liegen, d.h. $m_0(s)$ ist eine Multimenge über $C(s)$. Jede Kante e ist mit einem Ausdruck $E(e)$ über einer Menge von Variablen beschriftet, welche einen festgelegten Datentyp (gegeben durch eine Farbmenge) haben. Werden Variablen v eines Ausdrucks $E(e)$ durch Farben $b(v)$ der entsprechenden Farbmenge evaluiert[3], so ergibt $E(e)$ eine Multimenge von Farben $E(e) < b >$. Transitionen t können mit Bedingungen $G(t)$ beschriftet sein. Auch deren Variablen haben eine Farbmenge als Datentyp und können durch eine Funktion b an Farben gebunden werden. Durch eine solche Bindung evaluiert $G(t)$ zu *wahr* oder *falsch*. Eine Transition t kann in einer Markierung schalten, wenn eine Bindung b aller Variablen an Farben existiert, so dass die Transitionsbedingung zu *wahr evaluiert* und in jeder (Eingangs-)Stelle s mit $(s, t) \in F$ mindestens die durch $E(s, t) < b >$ beschriebenen Marken liegen. Schaltet t bzgl. einer solchen Bindung b, so werden diese Marken aus solchen Eingangs-Stellen entfernt, und in jeder (Ausgangs-)stelle s mit $(t, s) \in F$ werden die durch $E(t, s) < b >$ beschriebenen Marken hinzugefügt.

Wir werden Informationsbestandteile durch Marken geeigneter Farben modellieren.

2.2 Informationsbestandteile: Merkmal-Werte-Relationen

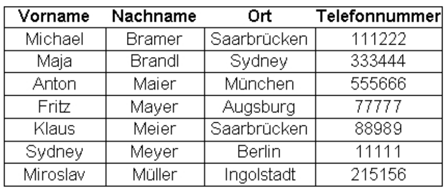
Grundsätzlich eignet sich das vorgestellte Modell ganz allgemein zur Beschreibung eines Dialogs zwischen Systemen, zwischen Menschen oder auch zwischen Mensch und Maschine, also generelle Informationsübertragung nicht nur per Sprache. O.B.d.A. wollen wir uns im Folgenden einen Mensch-Maschine-Dialog vorstellen, da sich dies vorteilhaft auf die Wahl geeigneter Abstraktionen auswirkt.

Im durchlaufenden Beispiel betrachten wir ein Mensch-Maschine-Dialogsystem, in dem der Benutzer des Systems einen Anruf mittels Spracheingabe tätigen möchte. Hierbei kann er die Nummer direkt angeben (falls er sie weiß), also z.B. “555666 anrufen” sagen. Oder er kann ihm bekannte Informationen zum gewünschten Anrufpartner, wie Vorname, Nachname und Ort, angeben. Er kann also sagen “Maja anrufen”. Zur Un-

[3] man sagt auch: die Variablen werden an Werte des entsprechenden Datentyps gebunden

terstützung des Systems existiert ein, in einer Datenbank gespeichertes, Telefonbuch mit Einträgen, die solche Informationen zur Verfügung stellen (Abb. 2).

Erste Aufgabe des Systems ist, übertragene Informationen zu verstehen. Üblicherweise wird eine Information (oder ein Informationsbestandteil) mit einer bestimmten Wahrscheinlichkeit verstanden. Kann der gewünschte Anrufpartner noch nicht mit ausreichender Sicherheit identifiziert werden, wird das System weitere spezifische Informationen erfragen. So ist es beispielsweise möglich, dass mit hoher Wahrscheinlichkeit ein bestimmter Nachname verstanden wurde, aber noch mehrere Personen mit diesem Nachnamen im Telefonbuch existieren. Diese könnten sich bzgl. des Wohnortes oder des Vornamens (oder beidem) unterscheiden lassen. Informationen werden so lange vom System “aufgesammelt” und “kombiniert”, bis der gewünschte Anrufpartner mit ausreichender Sicherheit feststeht.



Vorname	Nachname	Ort	Telefonnummer
Michael	Bramer	Saarbrücken	111222
Maja	Brandl	Sydney	333444
Anton	Maier	München	555666
Fritz	Mayer	Augsburg	77777
Klaus	Meier	Saarbrücken	88989
Sydney	Meyer	Berlin	11111
Miroslav	Müller	Ingolstadt	215156

Abbildung 2. Die Telefondatenbank des Sprachdialogsystems

Dieser Beschreibung zufolge können Informationen also verschiedenen Merkmalen wie *Vorname*, *Nachname* und *Ort* zugeordnet werden. Diese nennen wir Werte. Die Menge der Merkmale kann hierbei strukturiert sein, z.B. lassen sich Vor- und Nachname zum Merkmal *Name* zusammenfassen. Formal werden wir Informationen in sog. *Merkmal-Werte-Relationen* darstellen.

Definition 1 (Merkmal-Werte-Relation). *Gegeben seien zwei disjunkte endliche Mengen M (Merkmalmenge) und A (Menge atomarer Werte). Eine* Merkmal-Werte-Relation (MWR) über M und A *ist eine linkstotale, zyklenfreie Relation $R \subset M \times (M \cup A)$.*

Eine gewichtete *MWR ist ein Paar (R, π) bestehend aus einer MWR R und einer Menge $\pi = \{\pi_m \mid m \in M\}$ von Gewichten $\pi_m \in R^+$ auf den nicht-leeren Wertemengen $W(m) = \{w \in M \cup A \mid (m, w) \in R\}$ (d.h. $\pi_m : W(m) \to \mathbb{R}^+$). Handelt es sich bei π_m um Wahrscheinlichkeitsmasse, so spricht man von einer* stochastischen *MWR.*

Die MWR bzgl. eines Merkmals m notieren wir als $R_m := R|_{\{m\}\times(M\cup A)}$.

Zur Illustration der Zyklenfreiheit betrachten wir das Merkmal *Ziffernfolge*. Üblicherweise ist eine Ziffernfolge ein Element der unendlichen Menge $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}+$, die mit Hilfe geeigneter Rekursionen dargestellt werden kann (Abb. 3 links oben). Allerdings sind in einer speziellen Anwendung nur endlich viele Ziffernfolgen als Werte relevant, z.B. besitzen interne Telefonnummern meist eine vorgegebene Stellenzahl. Deshalb verzichten wir durch die Forderung der Zyklenfreiheit auf die Möglichkeit der Rekursion[4], was in unserer Situation einige Vorteile mit sich bringt.

Eine MWR hat, als Graph betrachtet, nicht notwendigerweise eine Baumstruktur (Abb. 3, rechts oben und links unten), dennoch kann man von *Wurzeln* (Merkmale, die nicht als Werte auftreten) und *Blättern* (atomare Werte, die tatsächlich als Werte auftre-

[4] In unserem Kontext könnte z.B. das Merkmal $m = \textit{Ziffernfolge}$ als Wertemenge $W(m)$ die Menge der internen Telefonnummern haben.

ten) sprechen. Falls nur *eine* Wurzel[5] vorhanden ist (Abb. 3, unten) und es sich um eine stochastische MWR handelt, kann man aus den Wahrscheinlichkeitsmassen $\{\pi_m\}_{m \in M}$ eindeutig ein Wahrscheinlichkeitsmaß π_R auf der Menge der Blätter konstruieren, und zwar ist für ein Blatt $b \in A$ seine Wahrscheinlichkeit $\pi_R(b)$ die Summe der Pfadwahrscheinlichkeiten der Pfade von der Wurzel zum Blatt b. Hierbei erhält man eine Pfadwahrscheinlichkeit durch Multiplikation der Wahrscheinlichkeiten entlang des Pfades. Falls die MWR eine Baumstruktur hat (Abb. 3, rechts unten), können aus einem Wahrscheinlichkeitsmaß π_R auf den Blättern auch umgekehrt die Wahrscheinlichkeitsmasse $\{\pi_m\}_{m \in M}$ berechnet werden.

Die in Abbildung 3 rechts unten dargestellte MWR benutzen wir in unserem durchlaufenden Beispiel eines Dialogs zur Anbahnung eines Anrufs zur Modellierung von Informationsbestandteilen. Blätter, die vom Merkmal *Datenbank* aus erreichbar sind, entsprechen grundsätzlich genau Datenbankeinträgen (vgl. Abb. 2). Die hier gezeigte MWR hat bereits Gewichte, die wir aber später erklären werden.

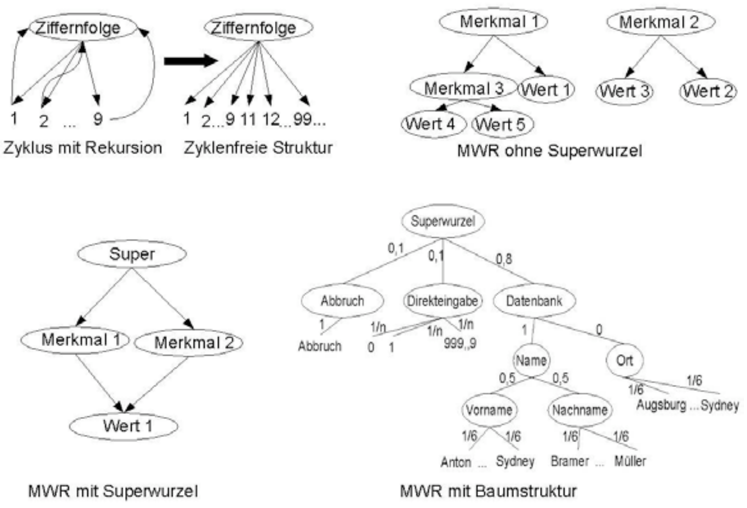


Abbildung 3. Zyklenfreiheit einer MWR und verschiedene Erscheinungsformen von Merkmal-Werte-Relationen.

2.3 Systembeschreibung

In diesem Abschnitt beschreiben wir das durch das Netz in Bild 4 modellierte Dialogsystem. Dazu bezeichnet M die Menge aller Merkmale und A die Menge aller atomaren Werte. Die Transitionen stellen bis dato abstrakte Funktionen dar, deren Ein- und Ausgabedaten durch Marken in den Stellen $p_1, \dots, p_6$ repräsentiert werden. Die durch die Stellen p_3, p_4, p_5 modellierten Daten repräsentieren hierbei gewichtete MWRen über M' und A' für geeignete Mengen $M' \subseteq M$ und $A' \subseteq A$. Dabei betrachten

[5] Gibt es in der MWR *genau ein* Merkmal, das nicht als Wert vorkommt, so bezeichnen wir dieses Merkmal als "Superwurzel".

wir o.B.d.A. nur MWRen mit Baumstruktur. Wir stellen eine solche gewichtete MWR $(R, \{\pi_m\}_{m \in M'})$ durch die Menge $\{(R_m, \pi_m) \mid m \in M'\}$ dar (Abb. 6). Eine Marke entspricht dann einem konkreten Paar (R_m, π_m). Die Stellen p_3, p_4, p_5 sind also mit der Farbmenge $C = \{(R_m, \pi_m) \mid m \in M\}$ beschriftet. Die Stellen p_1, p_2, p_6 haben einfachere Datenstrukturen, welche von der betrachteten Anwendung abhängen.

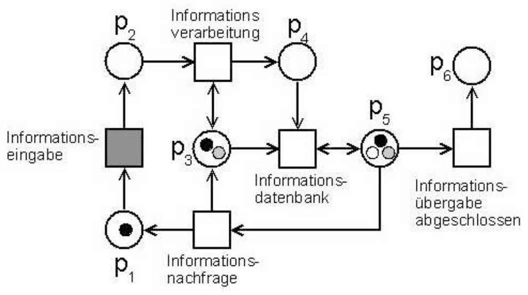


Abbildung 4. Abstraktes Petrinetz zur Informationsübertragung per Dialog.

Wie bereits erwähnt, betrachten wir einen Mensch-Maschine-Dialog. Wir konzentrieren uns dabei auf die Modellierung der Funktionalität der Maschine. Im dargestellten Netz beschreibt die graue Stelle *Informationseingabe* Funktionalität des Senders (also des Menschen), während das restliche Netz die Funktionalität des Empfängers (also der Maschine) repräsentiert. Ziel ist es, die durch die Informationseingabe generierte Information in p_2 so zu übertragen, dass der Empfänger diese in p_6 erhält. Dafür ist es notwendig den Zustand in p_5 so zu verändern, dass die Informationsübertragung abgeschlossen werden kann. Um diesen Vorgang zu verstehen, werden wir im Folgenden jede Transition mit Ein- und Ausgaben in Struktur beschreiben und interpretieren. Wir verifizieren dies anhand des durchlaufenden Beispiels eines Dialogs zur Anbahnung eines Anrufs.

Wir beginnen bei der *Informationsanforderung*.[6] Als Eingabe dieser Transition dienen Marken aus p_5. Die Stelle p_5 beinhaltet zwei voneinander unabhängige gewichtete MWRen. Die eine MWR (R^{IS}) beschreibt den aktuellen Informationsstatus des Empfängers durch Gewichtung g_i(Güte) von bereits erkannten Informationsbestandteilen i. Die andere MWR R^{bI} repräsentiert die noch benötigten Informationsbestandteile zur Vervollständigung der Informationsübertragung. Je größer das Gewicht eines Merkmals aus R^{bI} ist, desto dringender wird diese Information benötigt, um die Informationsübertragung erfolgreich abzuschließen. R^{bI} dient als Eingabe der Informationsanforderung, d.h. die Kantenbeschriftung ist derart, dass genau die den Merkmalen von R^{bI} entsprechenden Marken konsumiert werden.[7] Auf der Basis von R^{bI} wird eine entsprechende Informationsanfrage (an den Sender) in p_1 generiert (z.B. repräsentiert durch einen String). Diese Anfrage ist mit einer gewissen Erwartung an die nächste Informationseingabe durch den Sender verbunden. Grundsätzlich können Informationen nur dann verstanden werden, wenn eine gewisse Erwartung an den Inhalt der übertragenen Information existiert.[8] Eine solche Erwartung modellieren wir durch eine stocha-

[6] Die Informationsübermittlung mit der Nachfrage beginnen zu lassen widerspricht zwar dem Shannonschen Kommunikationsgedanken, aber diese Transition eignet sich am Besten zur schrittweisen Beschreibung des Systems. Dies lässt sich aber ohne weiteres rechtfertigen, indem man die initiale Nachfrage als Kommunaktionseröffnung interpretiert.

[7] In dieser Arbeit verzichten auf die formale Angabe von Kantenbeschriftungen.

[8] Dies ist sowohl beim Menschen als auch z.B. in existierenden Sprachdialogsystemen der Fall.

stische MWR mit Superwurzel R^{EH}, die wir *Erwartungshorizont* nennen. Die zu den Merkmalen von R^{EH} gehörenden Marken werden in p_3 generiert.

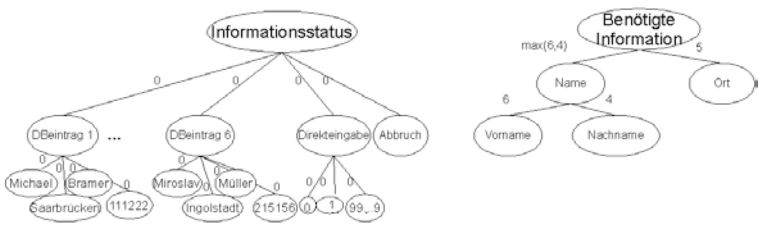


Abbildung 5. Links: R^{IS} - hier ist noch nichts erkannt worden. Rechts: R^{bI} - hier wird im Beispiel die Information als am notwendigsten eingestuft, welche die meisten (lautlichen) Unterschiede liefert.

Eine Analogie zur Informationsanforderung ist in unserem Beispiel der "Prompt" (Abb. 7). Im ersten Schritt entspricht R^{bI} einer Standardeinstellung und R^{IS} ist "leer", i.e. es ist $g_i = 0$ für alle i (Abb. 5). Aufgrund dieser Information wird der Benutzer (visuell, akustisch, textuell,...) aufgefordert den Namen des Teilnehmers zu nennen (p_1) und die entsprechende Erwartungshaltung (Erwartungshorizont in p_3) wird generiert (Abb. 6 und 3 rechts unten), wobei aber geringfügig berücksichtigt bleibt, dass der Benutzer den Anrufvorgang abbrechen oder die Nummer direkt angeben könnte. Der Ort wird nicht berücksichtigt, da $\pi_{Datenbank}(Ort) = 0$. In p_3 befinden sich jetzt also die Marken $(R_{Erwartung}, \pi_{Erwartung})$, $(R_{Abbruch}, \pi_{Abbruch})$, $(R_{Direkteingabe}, \pi_{Direkteingabe})$, $(R_{Datenbank}, \pi_{Datenbank})$, (R_{Name}, π_{Name}), $(R_{Vorname}, \pi_{Vorname})$ und $(R_{Nachname}, \pi_{Nachname})$.

Aufgrund der Informationsanfrage in p_1 liefert der Sender (Informationseingabe) eine Information in p_2 (z.B. in Form einer Lautfolge oder eines Strings). In unserem Beispiel antwortet der Mensch mit "Maja" (Abb. 8).

Die Informationsverarbeitung besitzt eine eingehende Information aus p_2 sowie den Erwartungshorizont R^{EH} aus p_3 als Eingabe. Die Informationsverarbeitung ordnet in Verbindung mit R^{EH} die eingehende Information einem Merkmal zu. R^{EH} wird anschließend wieder unverändert in p_3 zurückgelegt. Die Ausgabe in p_4 ist strukturell diesselbe stochastische MWR mit Superwurzel, die wir hier "Ergebnishorizont" R^{Erg} nennen, allerdings nun mit anderen Gewichten. Die Wahrscheinlichkeitsverteilung auf den Blättern von R^{Erg} repräsentiert das Erkennergebnis.

Der Erkenner (Abb. 9 links) erhält also die Information "Maja" als Sprachsignal. Gemäß des Erwartungshorizonts in p_3 wird dieses Signal als Name identifiziert, allerdings kann aufgrund der Lautähnlichkeit nicht festgestellt werden, ob nun tatsächlich der Vorname "Maja" oder der Nachname "Meier" gemeint war. Sicher ist aber, dass es sich weder um eine Ziffernfolge (also Direkteingabe der Telefonnummer) noch um den Befehl "Abbruch" handelt. Diese Information befindet sich nun als Ergebnishorizont in p_4 (Abb. 9 rechts).

Die Stellen p_3, p_4 und p_5 liefern nun die Eingabeparameter für die Informationsdatenbank. R^{EH} und R^{Erg} bilden in Kombination die Information nach der in der Datenbank gesucht werden soll. R^{IS} wird mit den Suchergebnissen aktualisiert und in p_5 zurückgelegt. Sollte das Suchergebnis nicht eindeutig sein, wird über die Informationsdatenbank eine MWR R^{bI} in p_5 generiert, welche die noch benötigten Informationen für den Abschluss der Informationsübertragung repräsentiert.

Nachdem der Erkenner mit einer hohen Wahrscheinlichkeit die Erwartung des nachgefragten Namens bestätigt hat, wird nun also in der Datenbank in den Vornamen nach "Maja" und in den Nachnamen nach "Meier" gesucht (bzw. nach Einträgen in Vor- und Nachname, die ausgesprochen der erkannten Lautfolge entsprechen)(Abb. 10). Beides existiert in der Datenbank und wird als Ergebnis in R^{IS} gespeichert, indem die Gleichverteilung der Gewichte durch die entsprechende Erkenngüte ersetzt werden. Leider ist die Suche daher nicht eindeutig und es erfolgt eine Auswertung, die das Merkmal "Ort" als hochgradig identifizierend einstuft und diese Information in R^{bI} einträgt (sehr hohe Gewichtung von "Ort", sehr geringe Gewichtung von "Name").

In p_5 befinden sich jetzt also aktualisierte MWRen R^{IS} und R^{bI}. Sofern der Informationsstatus vollständig ist, kann die Informationsübertragung abgeschlossen werden und die aktuelle Information, in Form der Marken R^{IS} wird an p_6 übertragen. Falls es sich allerdings um unvollständige Informationsbestandteile handelt, beginnt das System den eben erklärten Zyklus von vorne, wobei durch den Prompt auf Basis der gerade aktualisierten MWR R^{bI} wieder ein vollkommen neuer Erwartungshorizont erstellt wird.

Das Ende des ersten Durchlaufs in unserem Beispiel liefert uns die Information, dass der gewünschte Teilnehmer entweder den Vornamen "Maja" oder den Nachnamen "Meier" besitzt.[9] Diese Information ist nicht eindeutig und ermöglicht keinen Abschluss des Tasks. In diesem Fall wird also auf Basis von R^{bI} ein neuer Zyklus gestartet. Der Prompt generiert einen neuen Erwartungshorizont, welcher einen Ort als Erwartung widerspiegelt, worauf der Anrufer ihm "Sidney" nennt. Hätten wir nun keinen Erwartungshorizont, würde der Erkenner sowohl den Teilnehmer "Sidney Meyer" aus "Berlin" als auch "Maja Brandl" aus "Sydney" für möglich halten. Das Ergebnis wird durch den Erwartungshorizont relativiert und wir erhalten nach zwei Schritten "Maja Brandl" aus "Sidney" als eindeutig indentifizierten Anrufpartner und sind in der Lage den Task abzuschließen.

3 Ausblick

Ziel ist es, dieses abstrakte Modell in die Realität umzusetzen und die Transitionen zu implementieren. Nach Möglichkeit wollen wir hierfür weiter farbige Petrinetze verwenden. Als nächsten Schritt soll das vorgestellte Modell komplett mit Kanten- und evtl. Transitionbeschriftungen formalisiert werden.

Literatur

1. X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing*. Prentice Hall International, 2001.
2. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, volume 1-3 of *Monographs in Theoretical Computer Science*. Springer, 1992, 1994, 1997.
3. C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948. Continued in following volume.

[9] In diesem Fall sind die Gewichte g_i in R^{IS} aus Abbildung 5 verändert worden. Möglich wäre jetzt $g_{Vorname} = 1$ von DBeintrag2 und $g_{DBeintrag2} = 1$.

Appendix

Zur Illustration stellen wir hier einige ergänzende Bilder zur Verfügung.

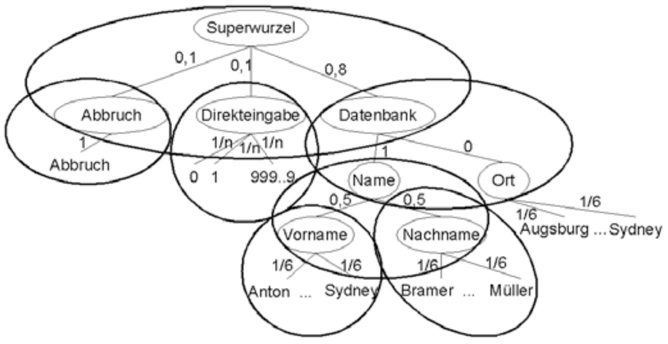


Abbildung 6. Jeder Kreis entspricht einer Marke (R_m, π_m) in p_3 in Abbildung 7

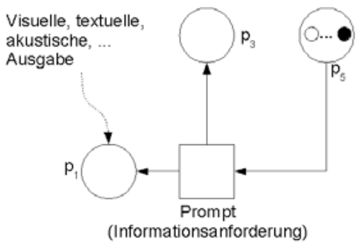

Visuelle, textuelle, akustische, ... Ausgabe
p3
p5
p1
Prompt
(Informationsanforderung)

Abbildung 7. Der Prompt generiert aufgrund von R^{bI} aus p_5 eine Anfrage in p_1 nach dem Namen und einen damit verbundenen Erwartungshorizont in p_3.

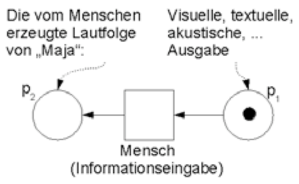

Die vom Menschen erzeugte Lautfolge von „Maja“:
Visuelle, textuelle, akustische, ... Ausgabe
p2
p1
Mensch (Informationseingabe)

Abbildung 8. Der Teilnehmer antwortet auf den Prompt mit “Maja”

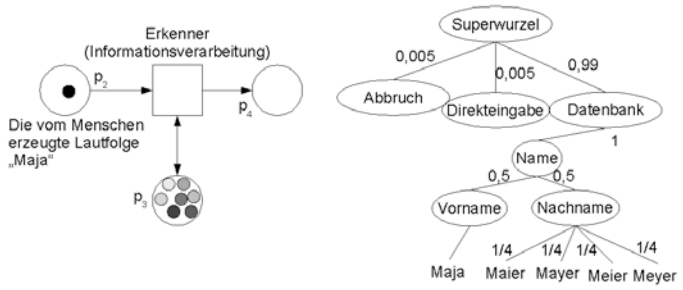


Abbildung 9. p_2 und p_3 dienen als Eingabe für den Erkenner. Dieser kann die Lautfolge “Maja” dem Merkmal “Name” zuordnen. “Abbruch” und “Direkteingabe” sind ausgeschlossen und deren Blätter werden im Ergebnishorizont in p_4 nicht berücksichtigt.

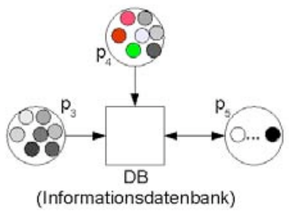


Abbildung 10. Die Datenbank erhält den Erwartungs- und Ergebnishorizont, sowie den Informationsstatus. Aufgrund dessen wird der Informationsstatus aktualisiert, sowie ggf. noch die benötigten Informationen generiert.

A Janus-Faced Net Component for the Prototyping of Open Systems

Matthias Wester-Ebbinghaus and Daniel Moldt

University of Hamburg, Department of Informatics
Vogt-Kölln-Straße 30, D-22527 Hamburg
http://www.informatik.uni-hamburg.de/TGI

Abstract. We introduce a Janus-faced reference net component that presents the basis for the recursive composition of complex systems from open system units. We particularly focus on the operational aspect of relating different levels of action at different system levels.

1 Introduction

We have presented various aspects of our organization-oriented software engineering approach SONAR/ORGAN [1–4] in previous contributions. SONAR (**S**elf-**O**rganizing **N**et **AR**chitecture) focusses on an exact mathematical body of rules and regulations for activities in an organizational position network. Orthgonally, ORGAN (**ORG**anizational **A**rchitecture **N**ets) provides a qualitative comprehension model for distinguishing different system levels according to distinctions between the (collective) organizational units studied at each level. In this paper, we focus on operationalizing the most central concept of ORGAN, namely building systems in terms of modular Janus-faced system units. This may be regarded as a prototypical basis that is open for incorporation of the SONAR rule set and orchestration according to the ORGAN architectural guidelines.

ORGAN rests on the universal model of an open and controlled system unit from Figure 1 that is applied at *all* system levels. We distinguish different internal system units (that are again instances of the universal model from Figure 1). Integration units together with operational units represent the "here and now" of the system unit in focus. The operational units are so to say the intrinsic units and carry out the system's primary activities. They are dependent on the integration units which offer a *technical frame* via intermediary, regulation, and optimisation services in the course of integration processes. The governance units represent the "there and then" of the system unit in focus. They offer a *strategic frame* via goal/strategy setting, boundary management, and transmitting their decisions to the other internal system units in the course of governance processes.

Each system unit is a *Janus-faced* entity. It "looks inwards" by embedding other system units and at the same time "looks outwards" by being itself embedded in other system units. Thus, besides the already mentioned internal (technical and strategic) frames, each system unit in focus is *externally framed* by surrounding system units to which the system unit in focus relates via periphery processes. To conclude, we take a recursive, self-similar nesting approach,

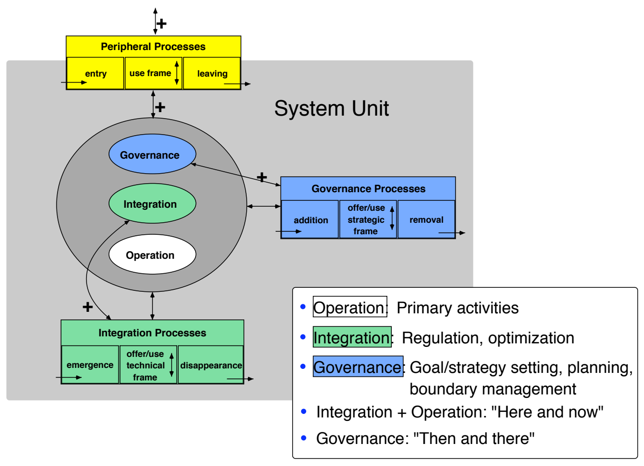

Peripheral Processes
entry
use frame
leaving
System Unit
Governance
Integration
Operation
Governance Processes
addition
offer/use strategic frame
removal
Integration Processes
emergence
offer/use technical frame
disappearance
+
+
+
+
Operation: Primary activities
Integration: Regulation, optimization
Governance: Goal/strategy setting, planning, boundary management
Integration + Operation: "Here and now"
Governance: "Then and there"

Fig. 1. Universal Model of an Open and Controlled System Unit

borrowed from Koestler's concept of a *holon* [5] that we extend with a generic reference model for control structures at each level. Thus, we arrive at a modular approach to comprehend *systems of systems*. Each system part may be regarded under a platform perspective and under a corporate agency perspective. All in all, this provides a conceptual basis to systematically study and implement different modes of coupling, both horizontally and vertically. In particular, we have conceptualised a reference architecture for multi-organization systems that exhibits four system levels: the departmental level, the organizational level, the level of organizational fields and the societal level.

In our previous contributions we have stressed the underlying Petri net semantics (specifically, reference net semantics [6]) of the model from Figure 1. However, the model remains rather abstract and we have omitted any real operationalisation details so far. In this paper, we have a look at one particular aspect of Petri net operationalisation in this context. We leave aside any details concerning a qualitative distinction between different system units and processes, how exactly system processes come into being and the addition, removal, migration or expansion of system units. Instead, we focus on one possible technical realization of relating different levels of action at different system levels with each other. We present a recursive approach that allows us to prototype open systems in a modular way, namely by addressing each system level and system unit in turn. Vertical as well as horizontal ties between system units are established via their inclusion in system processes through customisable generic interfaces.

2 Janus-Faced Net Component

Figure 2 displays a reference net component that reduces the various concepts from Figure 1 to internal system units, system processes and internal or peripheral actions of system units in the course of these processes. The component has the already mentioned Janus-faced character: It “looks inwards” by providing a platform for its embedded system units and at the same time “looks outwards” by being itself embedded inside other enclosing system units.

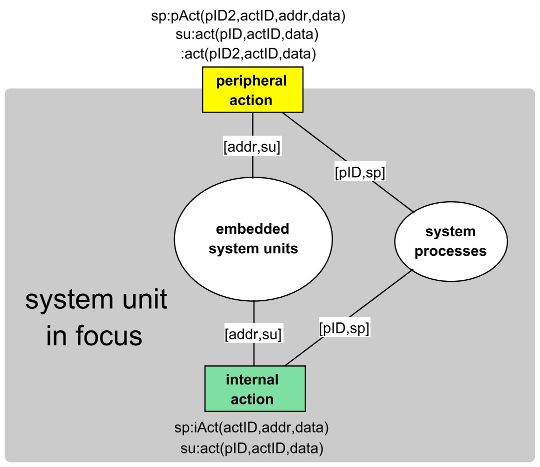


Fig. 2. Janus-Faced Net Component

Each internal action takes place in the course of some system process (sp) and is carried out by some internal system unit (su). It carries an action identifier (actID) that is unique in the context of the associated process. The corresponding system unit is identified by its address (addr) and is passed the identifier of the process (pID) in whose course the action is carried out. Finally, each action is associated with some data (data).

Peripheral actions are also carried out by internal system units. However, from the perspective of surrounding system units, they are carried out by the system unit in focus as a whole. Thus, we arrive at a technical understanding of *collective/corporate* action. As an additional argument, peripheral actions are not only associated with an internal system process but also with the corresponding identifier of the system process of the surrounding system unit (pID2).

We first have a look at a simple production process in Figure 3 with only internal actions. It can be read according to the common UML understanding: Places and Transitions on a vertical line represent the *life line* of some role while places and transitions on a horizontal line represent a *message exchange* between different role players. As we do not look at process instantiation here, we assume that the process net exists permanently (on the place system processes) and may be used for multiple concrete production scenarios. System unit addresses are associated with process roles and product identifiers dynamically at the first transitions of the corresponding life lines.

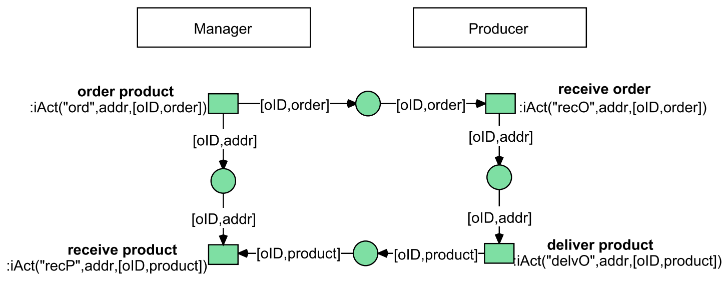


Fig. 3. Simple Production Process

3 Open System Prototyping

We now take a look at open systems by extending our modelling from the previous subsection with an additional system level that brings with it a collective level of action as can be seen in Figure 4. We have to decide, which are the *atoms*

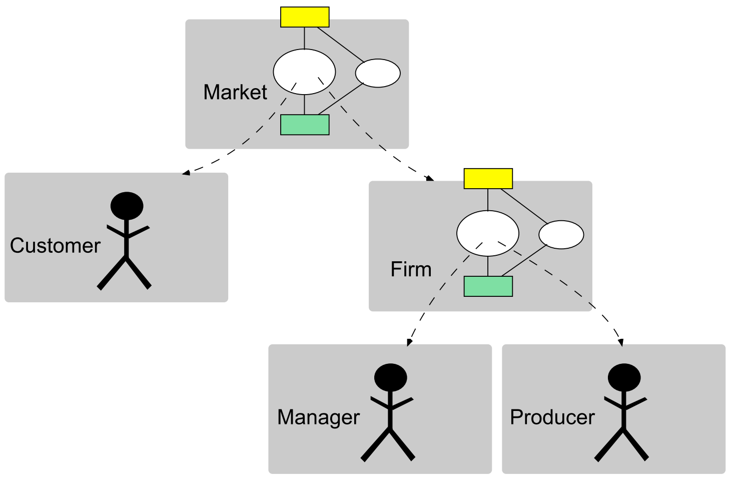
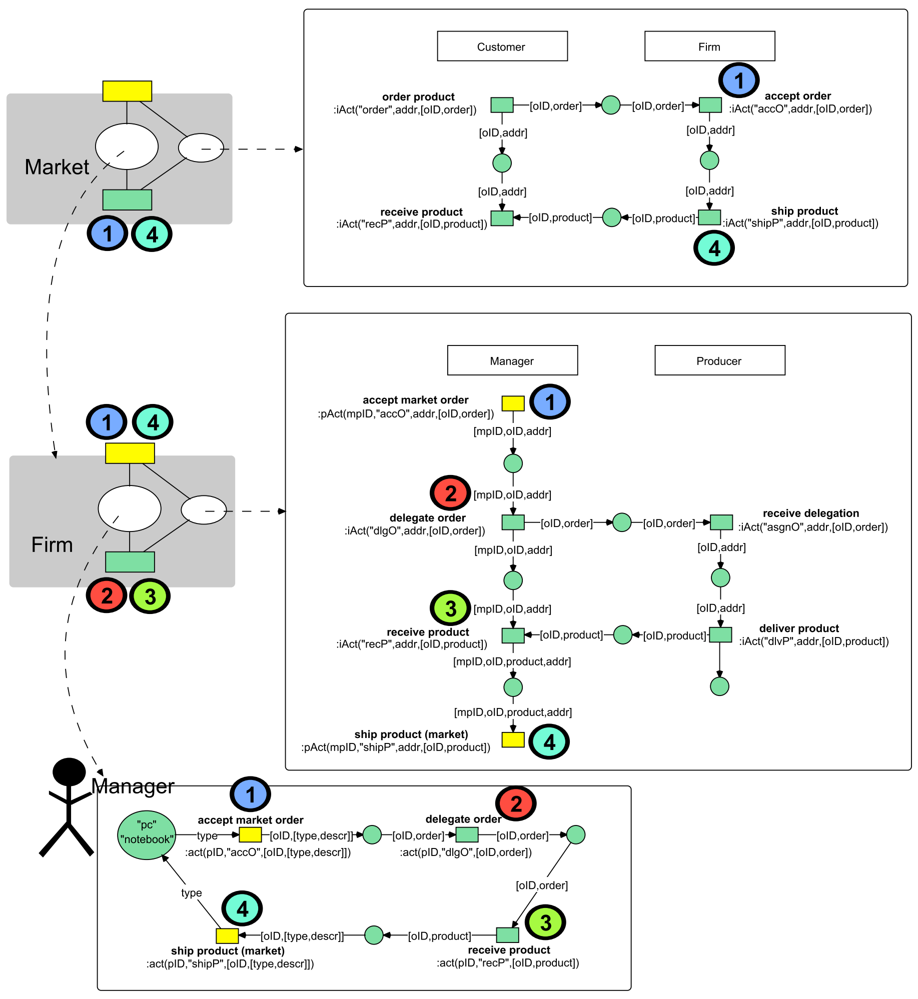


Fig. 4. System with Different Levels of Action

of our system, those parts from which all other activities eventually stem. These atoms are marked in Figure 4 as individuals. As they embed no internal system units themselves, they are not built according to the Janus-faced system unit from Figure 2.

We now have a look at how different levels of action interfere. As an example we take a look at the manager from the production firm that acts in turn on the market. Figure 5 displays how actions of the manager either only take place on the level of the firm or are additionally transformed to collective level actions of the firm itself on the market.



Market
Customer
Firm
order product
:iAct("order",addr,[oID,order])
[oID,order]
accept order
:iAct("accO",addr,[oID,order])
[oID,addr]
receive product
:iAct("recP",addr,[oID,product])
[oID,product]
ship product
:iAct("shipP",addr,[oID,product])
1
4
Firm
2
3
Manager
Producer
accept market order
:pAct(mpID,"accO",addr,[oID,order])
[mpID,oID,addr]
delegate order
:iAct("dlgO",addr,[oID,order])
receive delegation
:iAct("asgnO",addr,[oID,order])
receive product
:iAct("recP",addr,[oID,product])
deliver product
:iAct("dlvP",addr,[oID,product])
[mpID,oID,product,addr]
ship product (market)
:pAct(mpID,"shipP",addr,[oID,product])
Manager
"pc"
"notebook"
type
accept market order
:act(pID,"accO",[oID,[type,descr]])
[oID,[type,descr]]
delegate order
:act(pID,"dlgO",[oID,order])
[oID,order]
receive product
:act(pID,"recP",[oID,product])
[oID,product]
ship product (market)
:act(pID,"shipP",[oID,[type,descr]])

Fig. 5. Interplay between Different Levels of Action

– **Step 1:** The manager accepts a product order that occurs on the market. The manager accepts on behalf of the firm. Consequently, it acts on two system levels simultaneously: (1) at the firm level where a new production activity is initiated and (2) at the market level where the manager's action is transformed into a collective level action of the firm.

– **Steps 2 + 3:** The simple production scenario from Figure 3 can be found (slightly modified) as a substructure of the system process at the firm level in Figure 5. It still contains only (firm-)internal actions and in steps 2 and 3 the manager delegates the market order to some producer of the firm and afterwards receives the finished product.
– **Step 4:** The manager ships the finished product to the customer on the market. Just like in step 1, the manager acts both on the firm level (itself) and on the market level (its action transformed into a collective level one on behalf of the firm).

4 Outlook

We have presented a reference net operationalisation for combining different levels of action in complex systems that are hierarchically built up from open system units. This addresses one particular aspect of our previously published ORGAN-model for building software systems in the large as systems of systems.

Our operational approach rests on a generic Janus-faced net component that allows to recursively nest system levels inside each other. The generic interface of the component allows to regard each system unit without the need to bother with internal details of lower-level system units. For instance, in the example from Figure 5 it would easily be possible to expand the manager part into a complex system unit containing multiple managers (e.g. a salesman for market exchange management and a foreman for internal production management). This move would have no effect on the current process definition at the firm level. Consequently, our approach allows for the modular prototyping of hierarchically organized systems of systems that are composed of open system units.

References

1. Wester-Ebbinghaus, M., Moldt, D.: Structure in threes: Modelling organization-oriented software architectures built upon multi-agent systems. In: Proceedings of the 7th International Conference an Autonomous Agents and Multi-Agent Systems (AAMAS'2008). (2008) 1307–1311
2. Köhler, M.: A formal model of multi-agent organisations. Fundamenta Informaticae **79**(3–4) (2006) 415–430
3. Köhler, M., Wester-Ebbinghaus, M.: Closing the gap between organizational models and multi-agent system deployment. In: Multi-Agent Systems and Applications V. Volume 4696 of Lecture Notes in Computer Science., Springer-Verlag (2007) 307–309
4. Wester-Ebbinghaus, M., Moldt, D., Köhler, M.: From multi-agent to multi-organization systems: Utilizing middleware approaches. To appear, accepted paper for the 9th International Workshop on Engineering Societies in the Agents World (ESAW'2008) (2008)
5. Koestler, A.: The Ghost in the Machine. Henry Regnery Co. (1967)
6. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)

Proposal for Editing Workflows of a Distributed Software Development Environment

Kolja Markwardt, Daniel Moldt and Jan Ortmann

University of Hamburg, Department of Informatics,
Vogt-Kölln-Str. 30, D-22527 Hamburg
`http://www.informatik.uni-hamburg.de/TGI`

Abstract In distributed software development projects the different parties involved can be coordinated by the use of flexible workflow management systems (WfMS). Often the process cannot be defined completely in the beginning of a project or has to be adapted later on when conditions change.
In this paper the handling of workflow change in the agent-oriented POTATO system for distributed development will be presented. This includes the interaction of the different agents and their protocols as well as the mechanisms for ensuring the soundness of workflows even if they are changed during their execution.

1 Introduction

For distributed software development, the definition and enactment of processes with workflow management systems (WfMS) is an important means of structuring the interaction between the different participants. Often it can be important to modify these processes during the course of the project. This can be to reflect changes in the general conditions which require changes in the workflow. In other cases, it is not possible to completely specify the required process in the beginning of a project, so that some parts of the workflow can only be defined at a later stage. In ad-hoc workflows defined for one instantiation only, this is obviously more common than in production workflows, in both cases modifications can be necessary, though.

In these cases the workflow process definitions or even the running workflow instances need to be modified. Often it is difficult to decide whether a certain modification can be safely enacted on a workflow. Therefore instead of changing a running workflow changes are often only applied to a new instance, or special monitoring is required. Since workflows in the POTATO system (Process-Oriented Tool Agents for Team Organization) are specified with Petri nets, net-based methods can be used to ensure soundness of these modifications. In section 2 the POTATO system will be described with a focus on the process infrastructure used to enact workflows. Section 3 describes the methods for modification of workflows within the system. In section 2.4 the algorithms for checking the modified workflows for soundness are presented.

2 Workflows in Distributed Software Development

This paper describes the editing of workflow definitions in the context of the POTATO system for distributed software development. Therefore this section outlines the main properties of this system.

POTATO is an agent application built on the (Petri net based) MULAN/CAPA agent platform (see [7,2]). This agent platform itself is built using reference nets and uses the reference net editor/simulator RENEW[6] as its execution environment, which is implemented in Java.

The structure of POTATO is twofold: It has a tool-based organization that allows users of the system to equip their user agent (UA) with different kinds of tool agents (TA), in order to execute the tasks in the system they need to do. On the other hand a process infrastructure allows the execution of workflow processes in the system, connecting and integrating the different users.

2.1 User, Tool and Material Agents

The main goal of the POTATO system is to facilitate the work of different people working together to produce software. To achieve this, users can use different tools to manipulate materials, which are over the course of a project transformed into work results. This follows the notions of the tools and materials approach [10], applied to multi-agent systems to address distributed workplaces.

The main idea about the tool agent concept is that each user controls a user agent (UA), which can be enhanced by different tool agents (TA) (see [4,3]). The user agent provides basic functionality like a standard user interface and the possibility to discover and load new tool agents (tools).

Those tool agents can then plug into the user agents UI with their own UI parts, offering their functionality to the user. By choosing the specific set of tool agents, the user can tailor his workspace to his specific needs. A developer for example needs a completely different workplace then a tester or someone writing documentation.

Material agents (MA) are used to represent and encapsulate the materials or work objects that are currently worked on. Materials are manipulated by tools and can be created, deleted and moved between workplaces. Tools and materials populate the workspace of the user.

2.2 Process Infrastructure

A generic agent-based process infrastructure has been created (see [5]) and is used for POTATO. The process infrastructure offers the services of definition and execution of workflow processes in the development environment. It models a complete workflow management system (WfMS) using agent technology. This allows to make use of agent-based features, like distribution and mobility, so that the resulting WfMS is much more flexible than a normal stand-alone one. Within POTATO it is adapted to fit into the user/tool-agent structure. To organise the cooperation of different people working together on a project, workflow processes can be defined and enacted.

2.3 Workflow and its Soundness

In [8] workflow nets as a special form of Petri net are defined as well as soundness criteria. A test for checking workflow soundness is also given, which can be executed automatically using for example the Woflan tool [9].

As one of the most wanted properties during the execution of a workflow no deadlocks should occur nor should tokens be “lost” in the process. Therefore the notion of soundness has been defined. If a workflow net is started with a token on the start place, no matter which firing sequence occurs, it will always be possible to reach a marking in which only the end place is marked, and this is the only reachable marking in which the end place is marked. To check this property, the short-circuited net is constructed, by adding a transition to the net from the end place to the start place. Iff this net is live and bounded, the original workflow net is sound.[8]

There are different degrees of possible modifications, that can be handled

2.4 Keeping it Sound

differently. If the definition of a workflow is changed with no currently running instances or if the currently running instances are not to be changed, it suffices to check the new workflow definition for soundness, without any concern for the old definition.

The same is the case, if already running instances are concerned, but changes only occurr within subworkflows, which are not yet started. This is often the case if sections of a process are not specified when the execution begins, and only placeholder subworkflows are inserted to be defined later on. As long as the unspecified segments are workflows of their own, simply checking for soundness of the new definition is sufficient here, too.

It gets problematic however, if workflows have to be changed that already have running instances associated and those instances have to be converted to the new version. Since these instances can be in various states of execution, the standard method of checking for workflow soundness [8] cannot be applied directly.

Ensuring the Soundness of Workflow Modification A workflow definition is considered sound, if for every possible firing sequence it is possible to reach a state, in which only the final place is marked with a token. As mentioned above, for a normal workflow this can be ensured by constructing the sort-circuited net and checking it for the live and bounded properties.

In the case of a modified workflow, the structure of the net changes during the execution, therefore some special constructions are necessary to ensure the soundness.

For every place in the original workflow net a corresponding place in the modified net has to be defined, so that the transition to the new workflow can be executed and checked for soundness. To do this a modification net can be constructed as follows.

For a workflow net its modification net consists of all places, transitions and arcs from the original net as well as the modified net (disjointly united), connected by transitions from all places in the original net to the modified net.

With this construction it is possible to convert any instance of the original net to the new definition since all tokens are then accounted for. With the construction as seen in figure 1 it should also be possible to check the absence of deathlock possibilities during the transformation by checking the modification net for soundness. We will not try to formalize or proof this here though.

Example An abstract example for a Wf modification net can be seen in figure 1. The original workflow W_A consists of two parallel branches, one of which consists of two tasks. In W_B the other branch has got two parallel tasks. In t_{trans} the fact is accommodated, that no tokens must be left behind. Therefore even though one of the parallel tasks is deleted in the modified workflow, the corresponding tokens must be disposed of. Similarly, the task that is split up into two tasks in W_B requires two tokens to be generated in the target workflow.

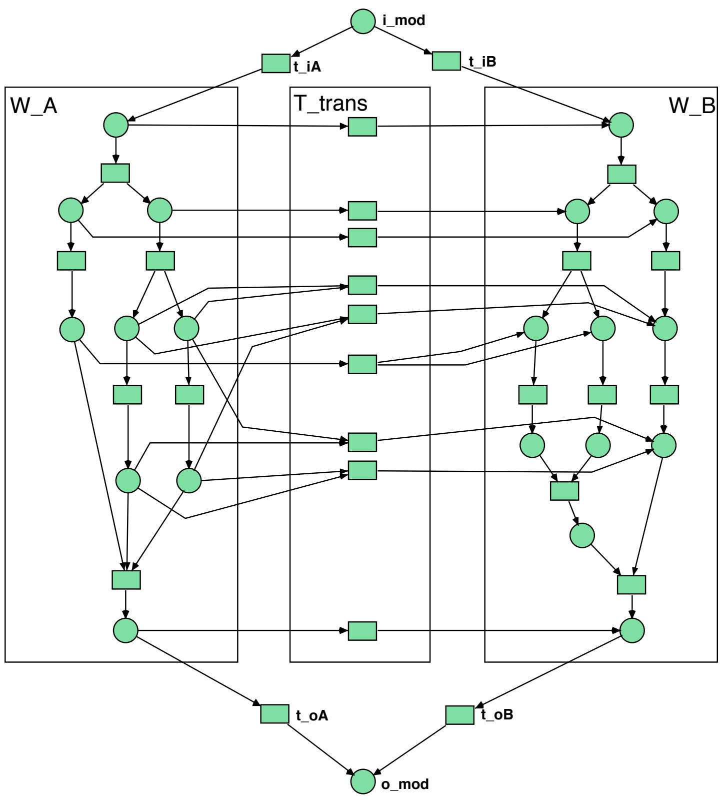


Figure 1. Workflow modification net

3 Modifying Workflows in the Potato System

In this section the mechanisms for modifying workflows are described. Since Potato is agent-oriented, the change process involves a couple of agents.

3.1 Involved Agents

Users with the appropriate authorization can edit workflows. To do so, they use a workflow edit tool agent via their user agent. This tool agent communicates with the workflow definition database agent to get the old workflow definition as a workflow definition material agent. This is then edited and the new version uploaded back to the definition database.

3.2 Adding New Workflows With the WF Edit Tool Agent

To add a new workflow definition, the net editing features of Renew can be used. Additional properties of the workflow can be defined in a special tool, like roles and rules for execution, workflow specific context data, tasks etc.

A material agent is then created and sent to the workflow definition database agent, where it is checked for soundness. This can not find all problems with a workflow, as many problems can occurr in the accompanying definitions of roles, participants, tasks etc. and not in the net structure itself, but at least some of the problems can be avoided this way.

3.3 Modifying Workflows and Their Definitions

If a workflow is to be modified, it must be decided how to handle already running instances of this workflow. If old instances are to be finished according to the old definition, the modification can be handled like a new workflow. The new version can then be added just like a completely new workflow.

If running instances are to be updated to the new definition however, care must be taken to migrate the processes correctly. In section 2.4 a simple algorithm is described to ensure soundness of workflow modifications. It is mandatory however, to specify the migration from the old to the new definition. For every place in the old workflow a place in the new workflow must be specified, so that all tokens can be moved over to the new definition.

In the editing process therefore a special mapping phase has to be added, in which this can be defined. By default it is sensible to assume, that all places existing in both versions of the workflow net are mapped to themselves, but it needs to be checked by the modifying user. If the workflow editing consists of a series of soundness-preserving transformations, each of these transformations could be assigned a default pattern of transformation, which can be adjusted by the user.

Then the new workflow definition along with the migration mapping is sent over to the workflow definition database agent, where it is verified. If verification is successful, the new version is saved as the new default for this workflow type. All workflow engines currently executing instances of the old definition must then be notified of the change and the migration mappings be applied.

4 Conclusion and Outlook

POTATO integrates the ideas and concepts from [1] and [4,3] to provide a framework. This shall allow for the support of workflows within a group collaborating in a distributed way. Here we proposed the use of user, tool and material agents, which cover specific roles within the application. They allow for easy editing of agent based workflows. Furthermore, we proposed to add formal checks on the workflows resp. their modification at runtime, based on traditonal techniques. The transfer of markings from one running instance of a workflow to another can e.g. be based on the places cuts.
In the long run it is planned to apply these concepts to our own software development process and environment. Therefore, the RENEW-IDE will be enhanced considerably.

References

1. Lawrence Cabac. Multi-agent system: A guiding metaphor for the organization of software development projects. In Petta Paolo, editor, *Proceedings of MATES'07*, volume 4687 of *LNCS*, pages 1–12, Leipzig, Germany, 2007. Springer.
2. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In *Agent-Oriented Software Engineering III: Revised Papers and Invited Contributions*, number 2585 in Lecture Notes in Computer Science, pages 59–72, Berlin Heidelberg New York, 2003.
3. Kolja Lehmann, Lawrence Cabac, Daniel Moldt, and Heiko Rölke. Towards a distributed tool platform based on mobile agents. In *Proceedings of MATES'05*, volume 3550 of *LNAI*, pages 179–190. Springer, September 2005.
4. Kolja Lehmann and Vanessa Markwardt. Proposal of an agent-based system for distributed software development. In Daniel Moldt, editor, *Third Workshop on Modelling of Objects, Components and Agents (MOCA 2004)*, pages 65–70, Aarhus, Denmark, October 2004.
5. Christine Reese, Jan Ortmann, Daniel Moldt, Sven Offermann, Kolja Lehmann, and Timo Carl. Architecture for distributed agent-based workflows. In B. Henderson-Sellers and M. Winikoff, editors, *Proceedings of AOIS'05*, pages 42–49, 2005.
6. RENEW – the reference net workshop homepage. `http://www.renew.de/`, 2008.
7. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
8. Wil M. P. van der Aalst. Verification of workflow nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, London, UK, 1997. Springer-Verlag.
9. H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
10. Heinz Züllighoven. *Object-Oriented Construction Handbook*. dpunkt Verlag, 2005.

Finding cost-efficient adapters

Christian Gierds[⋆]

Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin, Germany
gierds@informatik.hu-berlin.de

Abstract. When adapting services in a SOA environment, not only the validity of the adapter may be of importance, but also non-functional properties like the costs of the adapter. We introduce an approach for finding cost-efficient adapters based on the operating guideline, which characterizes all valid adapters for the given services.

1 Introduction

In the context of *Service-Oriented Architectures* (SOA) [1] composition of actually incompatible services, which have a well-defined interface in order to offer a special functionality, is highly demanded. A service of one organization may not have been designed to work together with a service of a different organization. But before changing one or even both of these services, an appropriate adapter may help to overcome the incompatibility. A (behavioral) adapter then acts between the two different services and controls their communication in such a way, that a certain set of functional properties like deadlock freedom or weak-termination is satisfied.

Especially in corporate environments costs like time, memory or money are relevant factors for components. So if a company decides to use an adapter, it may want that the overall runtime of the adapter stays below a certain limit in order to guarantee some real-time constraints or the costs for using third parties should be minimized. Besides this demand, the adapter still needs to be valid – the original goal to resolve incompatibilities must be maintained.

Our approach focuses on the minimization of the most expensive run of an adapter, meaning that for every other valid adapter the most expensive run is at least as expensive as for the calculated adapter.

The paper is structured as follows: In Sect. 3 we will describe the approach which covers both the validity and the cost optimization of adapters. Before it, we will introduce the basic formalisms in Sect. 2, namely open nets and operating guidelines. In the last section, we will summarize the obtained results and give an outlook on extensions of this approach.

2 Adapting services

An adapter is an artifact acting as mediator between services. This is necessary if the adapted services are incompatible regarding their interface or their behavior.

[⋆] Supported by German Research Foundation (DFG) under grant RE 834/18-1

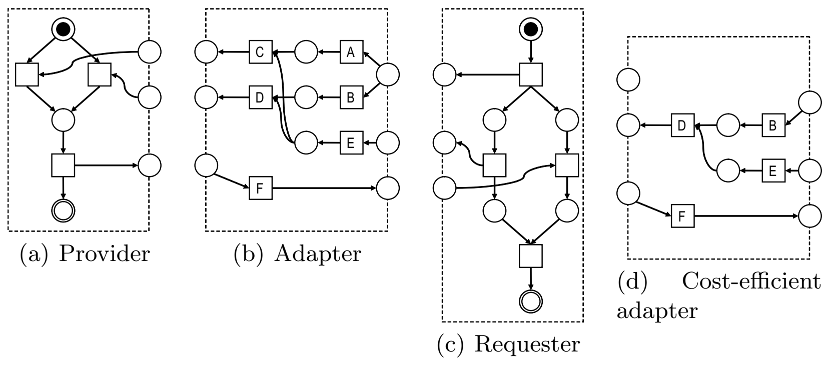


Fig. 1. Example

The adapter then should overcome these incompatibilities and guarantee a well behaved interaction of the services.

There are different approaches for adapters like [2–6] and recently also [7], to which we will refer. They mainly differ in the way, how elementary actions of an adapter are described and derived, and how the actual adapter is calculated.

In [7] a two-step approach is presented. First, for two given services P and R a *rewriter part* E (part of the final adapter) connects to the interfaces of P and R, provides transitions to manipulate messages based on a set of simple rules, and creates an interface for triggering these transitions. In the second step, a *controller part* C for the composition of $P \oplus E \oplus R$ is calculated such that certain properties like deadlock freedom etc. are ensured (see [8]). The suggested cost optimization in this paper is executed on C.

Open nets The used adapter approach is based on *open nets*, an extension of Petri nets, where distinguished places act as interface.

Definition 1. *The tuple* $(P, I, O, T, F, m_0, \Omega)$ *is called an* open net *iff*

- (P, T, F, m_0) *is a Petri net with a set of places* P*, a set of transitions* T*, a flow relation* $F \subseteq P \times T \cup T \times P$ *and an initial marking* $m_0 : P \to \mathbb{N}$*,*
- I *and* O *are disjoint sets of input* $I \subseteq P$ *and output* $O \subseteq P$ *places,* $I \cap O = \emptyset$*, and*
- Ω *is a set of final markings.*

Two open nets can be composed ($\oplus$) by merging equally named input and output places. The firing rule is equivalent to the one of regular Petri nets. A marking is not called a deadlock if it is included in the set of final markings.

The nets depicted in Fig. 1 are open nets. The places on the dashed border form the interface, all places belonging to the initial marking contain a black

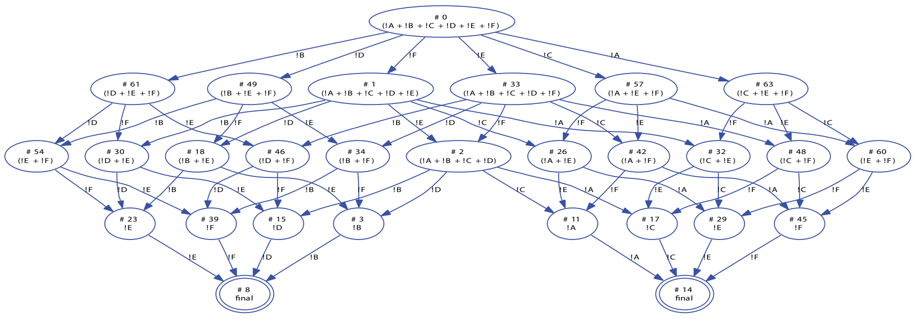


Fig. 2. Operating Guideline for Adapter in Fig. 1(b)

token and places belonging to a final marking are shown with a two line border. If the nets in Fig. 1 are composed as implied by the figures' alignment, the composition will be deadlock free.

Adapters The open nets P and R represented in Figs. 1(a) and 1(c) are not compatible, since the number of exchanged messages does not fit. Based on certain message transformation rules provided with the two services an adapter like Fig. 1(b) might be build based on the considerations in [7].

In our example let `A-F` be such rules that transform messages. Instead of executing these rules arbitrarily, we provide an interface such that these rules can be triggered by an controller. This controller will ensure certain properties like deadlock freedom, if wanted (see [11]). In the following, message exchange will be called an *event*.

Figure 2 shows such a controller. The graph, called *Operating Guideline*, is a labeled transition system, where each edge label represents an event, namely the sending or receiving of a message. Furthermore each node n has an annotation Φ over the labels of the edges leaving n. A Φ satisfying assignment β corresponds to a valid combination of edges, that have to be included in an controller, such that the controller is valid. In this paper we assume the operating guideline to be acyclic.

The controller part C can be transformed to an open net using the approach in [12], such that C and E can be composed to form a valid adapter $E \oplus C$.

3 Cost optimization

Looking at the possible adapters for P and R, besides some control structures we can mainly distinguish each single adapter by the transformation rules it can execute. It is legitimate to assume, that these transformation rules generate the costs in a corporate environment. In the simplest case, time is consumed to apply such a rule. In a more distributed scenario, such a transformation might be done by an external service provider, which will result in a fee which must be paid.

Just eliminating expensive transitions in the adapter is not an solution for finding cost-efficient adapters. When changing the adapter, we have to ensure that the correctness criterion like deadlock free communication is maintained. Therefore approaches like [9, 10] are not usable in this scenario, since we do not want to just calculate the cost for a service, but based on cost estimation build a service, in this case an adapter.

The operating guideline calculated during the adapter synthesis contains all information about legal adapters. Therefore an cost optimization should be done on this structure. There are two points we have to take care of, namely *i)* the most expensive run of the adapter shall be minimized and *ii)* the resulting structure must still be a valid adapter (thus, $P \oplus A \oplus R$ still must be deadlock free).

If we look at any given adapter A, it is still possible that A has different execution traces, i.e. sequence of events, depending on internal decisions in the services P and R.

The costs for one run of the adapter, a trace of events, is simply the sum of each applied message transformation. Thereby the cost function is a mapping of every transformation rule $r \in R$ to a natural number: $c : R \mapsto \mathbb{N}$.

Definition 2. *The* costs of a trace t *is the sum of its contained events:* $c(t) = \sum_{i=1}^{k} c(r_i)$ *for* $t = (r_1, \ldots, r_k)$.

We will focus on the question, which is the worst case, meaning, which are the highest costs we have to anticipate regarding the possible traces of A.

Definition 3. *The* costs of an adapter A *is the maximum costs over all traces of* A*:* $c(A) = \max_{t \in traces(A)} \sum_{i=1}^{k} c(t)$.

Based on Def. 3 our optimization goal is to find an adapter, whose costs are at most as expensive as for any other valid adapter.

Definition 4. *An adapter* A *is* cost-efficient *if for any other adapter* A' *yields* $c(A) \leq c(A')$.

The controller introduced in the previous section contains all information necessary for finding cost-efficient adapters. The annotations provide details about which other controllers are valid, and since every application of an transformation rule is communicated, it also contains all execution traces as a branching structure.

In order to find an cost-efficient adapter we will annotated each edge with a set of traces as follows. Given these edge annotations, we will compute an assignment for each node's annotation.

The costs incurred by using an edge is the maximum (the worst case) costs of the traces that are possible via this edge.

Definition 5. *The* costs of an edge e *is the maximum cost of its assigned traces:* $c(e) = \max\limits_{t \in traces(e)} c(t)$.

Given a satisfying assignment β for the annotation of a node n, β states which edges leaving n have to be included in a controller in order to be valid. The node's costs regarding β then is the maximum over the edges' costs (again, the worst case).

Definition 6. *The* costs for a node n and an assignment β_n *is the maximum cost of the edges e leaving n, which corresponding literal is set to true in* β_n*:*
$c_{\beta_n}(n) = \max\limits_{\beta_n(e)=true} c(e)$.

Since every satisfying assignment yields in a valid adapter, we choose an assignment, which results in the minimal costs for a node.

Definition 7. *The* costs for a node n *is the minimum costs over all assignments* β_n *satisfying n's annotation* Φ*:* $c(n) = \min\limits_{\Phi(\beta_n)=true} c_{\beta_n}(n)$.
The assignment β_n*, which minimizes* $c(n)$ *is called the* minimal β_n.

Given the previous definitions the following algorithm will calculate a cost-efficient adapter.

Algorithm Let P and R be two open nets, E a partial adapter, and C the operating guideline for $P \oplus E \oplus R$. Then an *cost-efficient adapter* can be found as follows:

Initially all edges have no traces assigned, and all nodes have infinite costs, except for the leaf nodes (without successor), which have costs 0 (resulting from Def. 7). Then, as long as there are nodes with infinite costs, pick such a node n, so that each successor n' of n has finite costs $c(n') < \infty$. Assign for each edge $e = (n, n')$ the set of traces according to the minimal $\beta_{n'}$ of n': $traces(e) = \{label(e) + t' \mid t' \in traces(e'), \beta_{n'}(e') = true\}$ (meaning: each new trace starts with $label(e)$ followed by the events in the traces of e'). Afterwards the costs $c(n)$ can be calculated.
The costs for resulting adapter $A = E \oplus C$ are the costs of C's root node.

Since we assume the operating guideline to be finite and acyclic, it can be easily seen, that the suggested algorithm will terminate, and all nodes will have finite costs. Furthermore we gain a valid controller (implied by the found minimal assignments), which minimizes the costs.

Theorem 1. *The provided algorithm finishes with a controller* C *such that* $A = E \oplus C$ *is a cost-efficient adapter for* P *and* R*.*

Proof. (Sketch.) This result yields mainly due to Def. 7. The adapter A is valid, since for each node n of the controller, the minimal β_n satisfies n's annotation (see [8] for details). Assume A' is another valid adapter with less costs $c(A') < c(A)$. Since both A and A' are derived from C there exists a node n included in both adapters, but differing in the minimal β_n, meaning $c_{A'}(n) < c_A(n)$, which contradicts Def. 7. Therefore the found adapter is valid and cost-efficient.

4 Summary

We have seen an approach which calculates a cost-efficient adapter based on an annotated graph which acts as controller for the application of the message transformation rules. By finding optimal assignments to the nodes' annotations we obtain a smaller, but valid adapter, where expensive runs can be excluded.

The time for finding such an adapter is exponential in the number of applicable message transformation rules in the worst case, since for every node all satisfying assignments must be checked. Nevertheless in most cases the approach should yield the result in a reasonable time, since the annotations are normally short and therefore quickly to be checked. In order to show its feasibility, this algorithm shall be implemented and checked with real-world examples.

As extension of this approach probabilities for the occurrence of events shall be introduced, so that the *average* costs of an adapter can be calculated. This extension would also allow to lift the optimization to cyclic controllers. Although cyclic adapters are finite, the have however infinite traces and therefore infinite costs.

References

1. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (July 2007)
2. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. Science of Computer Programming **61** (2006) 136–151
3. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R., Toumani, F.: Developing Adapters for Web Services Integration. In: Proc. CAiSE. Volume 3520 of LNCS. (2005) 415–429
4. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74**(1) (2005) 45–54
5. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing Web Service Choreographies. Electr. Notes Theor. Comput. Sci. **105** (2004) 73–94
6. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 65–80
7. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (August 2008)
8. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
9. Hee, K.M.v., Sidorova, N., Stahl, C., Verbeek, H.M.W.: A price of service in a compositional SOA framework. Computer Science Report 07/16, Technische Universiteit Eindhoven, The Netherlands (jul 2007)
10. vom Brocke, J., Lindner, M.A.: Service portfolio measurement: a framework for evaluating the financial consequences of out-tasking decisions. In: ICSOC. (2004) 203–211
11. Schmidt, K.: Controllability of Open Workflow Nets. In: Enterprise Modelling and Information Systems Architectures. Volume P-75 of LNI. (2005) 236–249
12. Badouel, E., Darondeau, P.: Theory of Regions. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS., Springer-Verlag (1996) 529–586

Transient Analysis Of Stochastic Petri Nets With Interval Decision Diagrams

Martin Schwarick
ms@informatik.tu-cottbus.de

Brandenburg University of Technology Cottbus, Germany

Abstract. This paper presents an Interval Decision Diagram (IDD) based approach to realize symbolically transient analysis of Continuous Time Markov Chains (CTMC) which are derived from stochastic Petri nets. Matrix-vector and vector-matrix multiplication are the major tasks when doing exact analysis of CTMCs. We sketch a simple algorithm which uses explicitly the Petri net structure and offers the opportunity of parallelization. We present results computed with our first prototype implementation.

1 Motivation

Stochastic Petri nets are a natural way to model biochemical networks, where token values are interpreted as levels of concentration [1]. A stochastic Petri net's semantics is a CTMC which can be analysed applying steady-state and transient analysis [2] or CSL [3] model checking. The tool of choice for these purposes is often the probabilistic model checker PRISM [4], which seems to represent the current state of the art. The description of stochastic Petri nets can be translated into the PRISM language, as done in [1]. To face the problem of state space explosion PRISM uses an engine based on Multi Terminal Binary Decision Diagrams (MTBDD) and symbolically performs analysis.
Using PRISM's MTBDD based approach in the context of stochastic Petri nets with a level semantics has several drawbacks; prior knowledge about boundedness of each place is required. A place which can carry up to k token must be represented by $|ld(k)|$ MTBDD variables. This results in an overhead in computation time and memory. Since a token represents a concentration level increasing the accuracy of analysis implies an increase of the possible number of tokens on places. PRISM creates an MTBDD which represents the entire CTMC. Therefore it is necessary to double the number of MTBDD variables. A further drawback occurs if the CTMC contains many different rate values, since the number of terminal nodes in the MTBDD equals this amount.
A. Tovchigrechko introduced in [6] very efficient algorithms for the state space based analysis of bounded Petri nets using IDDs. We combine the ideas and algorithms in [4][5][6] and use a slightly augmented form of IDDs to realize transient analysis of stochastic Petri nets. In section 3 we will sketch how it works. But before that we will briefly recall the most important concepts.

2 Preliminaries

In a stochastic Petri net an exponentially distributed firing rate is associated to each transition which is generally defined by a state-dependent hazard function. Since we consider mass action kinetics [1] this hazard function is defined as the product of a specific constant and the token values of the transition's preplaces of the state. The semantics of a stochastic Petri net is a CTMC which can be seen as a graph isomorphic to the reachability graph of the underlying Petri net, but state transitions are labeled with the firing rates. In general, CTMCs are represented as very sparse matrices indexed by states, which entries are real valued rates. Transient analysis determines for each state how probable it is to be in it at a certain time point. An established technique to realize transient analysis of CTMCs is the uniformization method [2]. Its basic operation is vector-matrix multiplication which must be done for a certain number of iterations.
Faced with the state space explosion problem, it is not worth thinking about implementing vector-matrix multiplication explicitly whereby the matrix and the vector are indexed by states.
In our approach the set of states is represented by an Interval Decision Diagram. We compute all needed data at each iteration anew from one augmented IDD representing the reachable states. That is the main difference to PRISM's approach, where the CTMC's state space and its rate matrix are represented symbolically by a BDD and a MTBDD.
We will give a brief and informal introduction to IDDs:

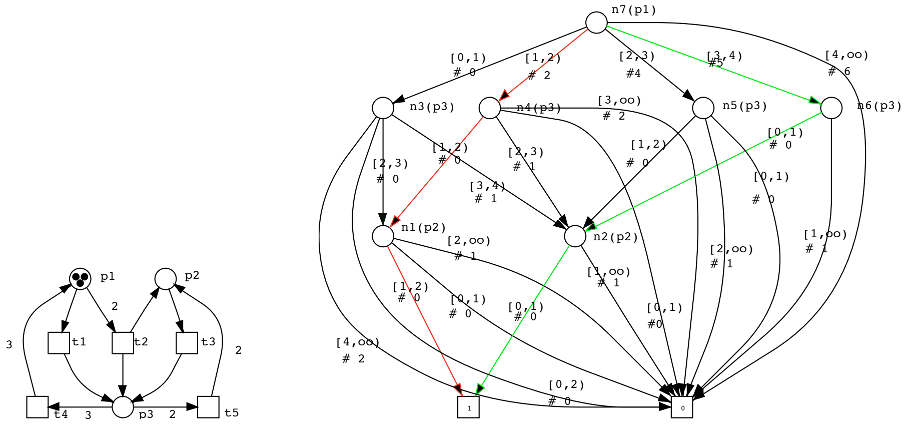


Fig. 1: A Petri net and the IDD RS representing its reachable states. The path $n7 \xrightarrow{3} n6 \xrightarrow{0} n2 \xrightarrow{0} 1$ represents state $m \equiv (p1 : 3, p2 : 0, p3 : 0)$. The path $n7 \xrightarrow{1} n4 \xrightarrow{1} n1 \xrightarrow{1} 1$ represents state $m' \equiv (p1 : 1, p2 : 1, p3 : 1)$ which can be reached from m by firing transition $t2$. Edges are labeled with an interval and the additional index data.

An IDD is a rooted, directed and acyclic graph which nodes can have any number of outgoing edges. Each edge is labeled with a left closed and right open interval on $\mathbb{N}$. The intervals of the outgoing edges of an IDD node define a partition of $\mathbb{N}$. There are two nodes without outgoing edges: the terminal nodes, labeled with ONE and ZERO. To each IDD node a variable is associated, in our context a place of the stochastic Petri net. We assume that on each way from the root to a terminal node, the variables occur in the same order. As for BDDs the variable ordering influences the IDD size. Furthermore we assume, that the IDD does not contain isomorphic subgraphs. A sequence of IDD nodes considering connecting edges reaching the ONE-terminal node represents a set of states. We will denote a path as such a sequence while chosing exactly one value from the interval of the occuring edges.
In the following section we will present an algorithm which performs vector-matrix (or vice versa) multiplication, whereby the matrix is defined by the reachable states of a stochastic Petri net, using only the IDD represenation and the net structure.

3 Multiplication by traversing

To realize a matrix-vector or a vector-matrix multiplication, whereby the matrix and the vector are indexed by states, we need a mapping from states to indices. The depth first search traversation of an IDD induces a lexicographic order of its represented states. Since a state is an unique path to the ONE-terminal node we must store some information for each outgoing edge which enables the index computation. For each edge we store the number of lexicographic smaller states, which can be reached over all its previous sibling edges of the respective node (See Fig. 1). We can also determine the number of states, which can be reached over an arbitrary edge.
The basic concept of our algorithm is to traverse for each transition t of the stochastic Petri net the IDD ES_t representing its *enabling states*. For each path in ES_t the IDD RS respresenting the *reachable states* contains a respective path. We can easily determine the lexicographic index for the associated state m using the additional index data during traversation. Since m is an element of ES_t there exists a path in RS, which represents the state m', reached by firing of transition t in m. While traversing the IDD ES_t we track the paths for m and m' in RS and compute their indices considering all reachable states. Each time the ONE-terminal has been reached we extract the indices of a matrix entry. Furthermore we must determine the associated rate value. Considering mass action kinetics implies to multiply the present rate with the current element of an expanded interval if the current IDD node is related to a preplace of t.
The resulting recursive algorithm below should be self-explanatory. All used functions can be implemented very efficiently. The function $getWeight(p : Place, t : Transition)$ returns the token change, the firing of t causes on p. If p is a

preplace for instance, the return value would be negative.

```
var
     transition, t : Transition;
     j: int;
procedure traverse (IDD_Node root, IDD_Node src, IDD_Node dest,
                     src_index int, dest_index int, rate double)

  begin
     if root = ONE then
       //e.g. vector-matrix r=v*M :
       //r[src_index] = v[dest_index]*rate
       //rate is M[src_index][dest_index]
       processData(src_index, dest_index, rate);
       return;
     end //if
     place: Place;
     rate2: double;
     value, value2, src_index2, dest_index2, i: int;
     src2, dest2: IDD_NODE;
     edge: Edge;
     place:= root.correspondingPlace();
     for 0 <= i < root.edges() do
       edge := root.edge(i);
       if edge.node() != ZERO then
         while value < edge.upperBound() do
           value2:= value;
           rate2:= rate;
           if isPrePlace(place, transition ) then
             rate2: = rate * value;
           end //if
           value2:= value + getWeight(place, transition);
           src2:= getChild(src, value);
           dest2:= getChild(dest, value2);
           src_index2:= src_index + smallerStates(src, value);
           dest_index2:= dest_index + smallerStates(dest, value2);
           traverse(edge.node(), src2, dest2,
                     src_index2, dest_index2, rate2);
           value:= value + 1;
         end //while
       end //if
     end //for
   end. // traverse_

   /* the following program code can be parallelized*/
   for 0 <= j < SPN.transitions() do
      t = SPN.getTransition(j);
      traverse(ESt.root, RS.root, RS.root, 0, 0, t.rate);
   end //for
```

As for every implementation of decision diagrams, efficiency depends on considering redundancies. In general nodes on inner IDD levels will be visited many times. Subpaths beginning in these nodes will be traversed each time anew. Like in [4] we set a certain IDD level and cache index and rate information for each of its nodes about all paths containg these IDD nodes. For shortage of space we must omit further details considering used data structures. Each time a node of this cache level has been reached, only the cached data must be processed. The remaining problem is to find an adequate level. Moving the cache level towards the root speeds up the computation at the cost of an increased memory consumption as can be seen in Table 1. We hope to find good heuristics based on the IDD structure and the Petri net structure.
This traversation algorithm can be applied concurrently for more than one transition. We have to care about synchronization of write access to the result vector only. Currently we realize this synchronization by allocating a result vector for each thread which performs traversation. When the traversation for all transitions is finished, result data must be gathered before the next iteration starts.

4 Results

We now present results obtained with our prototype, which is based on an IDD implementation of A. Tovchigrechko. Our biochemical model is the extended ERK pathway from [1]. The test system is a Dual Core Intel Xeon with 2,1 GHz and 2 GB main memory running a 64 Bit Linux. We made transient analysis for one second for the eight level version. The CTMC has 6,110,643 states and 78,948,888 transitions. The transient analysis requires 218 iterations. We compared the time per iteration and the memory usage obtained by using our tool on one and two cores with PRISM 3.2 (hybrid engine) as can be seen in Table 1. Currently our implementation requires significant more memory than PRISM. This is in dept to our current cache data implementation and the synchronization technique. Table 1 also underlines the impact of the cache level to iteration time and memory usage.

	idd transient						PRISM 3.2	
number of cores	1			2			1	
cache level	10	8	6	10	8	6	19	55
time per iteration (sec)	1.29	1.90	5.43	1.03	1.26	2.98	2.53	1.22
memory (MB)	534	408	393	581	455	440	251	323

Table 1: For the CTMC representation of this model PRISM constructs a MTBDD with 66 variables (levels). The IDD representing the state space has 22 levels. In both cases the level counter starts above the terminal level with zero and increases towards the root level. We set the cache level for our tool to 10, 8 and 6. PRISM sets the cache level for this model to 19. To increase the performance we run PRISM with different cache levels. Setting it to e.g. 55 halves the time per iteration.

5 Future Work

In the near future a bulk of work has to be done to enhance functionality and performance of our prototype.
functionality: As introduced in [7] and realized in PRISM we want to implement CSL model checking.
Presently only mass action kinetics are implemented. In the future our tool should handle arbitrary hazard functions as they can be specified with our Petri net editor Snoopy [8].
performance: One example for a performance improvement is transition grouping. Instead of one traversation for each transition we could group several transitions together and traverse the IDD for this transition group. Doing so should reduce the traversation effort and should have an effect like loop blocking resulting in better usage of the CPU's cache memory. First experiments provided promising results.
Moreover memory requirements must be reduced. To simplify synchronization each thread gets currently an own result vector of type double to store intermediate data. We will look for a better approach like Compare and Set (CAS) to get by with only one result vector.
For the time being we use multiple cores sharing common main memory. We are going to analyze whether our approach could be applicable in an environment with distributed memory.

References

1. Gilbert D., Heiner M., Lehrack S.: A unifying framework for modelling and analysing biochemical pathways using Petri nets, Proc. 5th International Conference on Computational Methods in Systems Biology (CMSB 2007), Edinburgh, September, Springer LNCS/LNBI 4695, pp. 200-216 (2007)
2. Stewart W. J.: Introduction to the Numerical Solution of Markov Chains. Princeton (1994)
3. Aziz A., Sanwal K., Singhal V., Brayton R.: Verifying Continuous Time Markov Chains. Proc. 8th International Conference on Computer Aided Verification (CAV96), Springer, pp. 269–276 (1996)
4. Parker D.: Implementation of symbolic model checking of probabilistic systems. University of Bermingham, PhD thesis (2002)
5. Miner A.S., Ciardo G.: A data structure for the efficient solution of GSPN. College of William and Mary Williamsburg (1999)
6. Tovchigrechko A.: Efficient symbolic analysis of bounded Petri Nets using Interval Decision Diagrams. Brandenburgische Technische Universität Cottbus, PhD thesis in press (2006)
7. Baier C., Haverkort B., Hermanns H., Katoen J.-P.: Model checking continuous-time Markov chains by transient analysis. Proc. 12th International Conference on Computer Aided Verification (CAV00), Springer, pp. 358–372 (2000)
8. Heiner M., Richter R., Schwarick M.: Snoopy - A Tool to Design and Animate/Simulate Graph-Based Formalisms. Proc. International Workshop on Petri Nets Tools and APplications (PNTAP 2008, associated to SIMUTools 2008), Marseille, ACM digital library (2008)

On synthesizing service behavior that is aware of semantical constraints

Karsten Wolf

Universität Rostock

Abstract. Without taking care of the semantics of messages, every message is an isolated entity that can be created and sent at will. This leads to anomalies like a synthesized service that sends a filled form before having received the empty form. In this paper we pick up ideas from adapter synthesis for taking care of semantical constraints and develop them into two directions. First, we show that the approach taken for adapter synthesis can be applied to synthesis of services in general. Second, we argue that the taken approach is in a certain sense complete.

1 Introduction

We synthesize service behavior for several purposes. First, we can show that it is possible to interact correctly with a service by constructing a fitting service behavior [1–3]. This approach can be used as a sanity check for given services. Second, we can try to exploit the canonicity of the computed behavior and use it for characterizing all correctly interacting partners of a service [4]. Third, we can transform the computed behavior into executable code that can be executed as a particular partner of the service.

An instance of the latter approach is the synthesis of a behavioral adapter A between two given services P and R. At first glance, A is not much more than a correctly interacting partner of a disjoint composition of P and R. However, this general setting permits a number of anomalies with arise from the fact that behavioral approaches typically abstract from the content of exchanged messages. In plain language, messages are named a, b, c, $\ldots$ without taking care of whether they denote the submission of a password, an address, a simple acknowledgement, a real item (sold book), or anything else. This simplification leads to anomalies like the synthesis of an adapter that can "invent" a password or forward two copies of a received (real!) book. Having observed this, virtually all approaches to behavioral adapter synthesis [5–11] start with some kind of specification that expresses appropriate constraints for the activities that can be performed on certain messages. Although different in technical detail, the specifications all express more or less the same class of constraints. This class of constraints can thus be considered as mature.

In this note, we demonstrate that the approach taken in adapter synthesis extends to the synthesis of service behavior in general. That is, we can avoid semantic anomalies in any kind of synthesis of services by taking into account an appropriate specification of semantical constraints. For instance, we can suppress synthesis of a service that sends a message containing a session identifier before having received this identifier. We can avoid sending a signed contract before having received the unsigned version thereof. There are many other examples of constraints that are imposed by the semantics of message contents.

As a second contribution, we argue about completeness of our approach. We claim that (under a few technical limitations like boundedness) we can synthesize a correctly interacting partner whenever one exists. To this end, we propose a definition for "arbitrary partner that respects given semantical constraints" and then show our completeness result using this definition.

2 Services

We model services as open nets, i.e. as place/transition Petri nets with an initial marking, a set of final markings, and a set of places serving as interface. Initial and final markings have no tokens on interface places. We require further that final markings do not enable any transition (although transition may become enabled by putting tokens on input places). The interface is divided into input and output places and we require that no transition takes tokens from an output place an no transitions puts tokens on an input place.

There exist translations from industrial languages like WS-BPEL into open nets [12] and vice versa [13] which proves the suitability of open nets for modeling services. Open nets are composed by merging equally named interface places (an input place of one service with an output place of the other one). The merged places are then removed from the interface. Initial and final markings are composed canonically (remember that we require them to have no tokens on interface places). We denote the composition of two open nets P and R by $P \oplus R$.

Composition may lead to an open net with empty interface which we call closed net.

A closed net is deadlock-free iff all markings without enabled transitions are final. An open net P is controllable iff there exists an open net R such that their composition $P \oplus R$ is a deadlock-free closed net.

It is just one possibility to use deadlock freedom as the underlying property for controllability. Other requirements could include livelock freedom or any other desired property. Deadlock freedom is, however, the most prominent property discussed in the context of web services.

3 Synthesis of service behavior

Controllability of an open net P is decidable [1, 2] if two conditions are satisfied. First, the inner of P (the net obtained by removing the interface of P) must be bounded and second, we restrict the set of considered partners to those R where the composition $P \oplus R$ yields k-bounded (now merged) interface places, for some a priori given k. In effect, the composition of considered nets are finite state systems. In absence of the first restriction, controllability becomes undecidable [14] even in presence of the second one. If only the second condition is dropped, decidability of controllability is unknown.

In presence of the mentioned conditions, controllability can be decided by synthesizing (the state space of) a canonical R as required by the definition of controllability. The resulting state space (a kind of automaton) can be transformed into a Petri net using standard approaches [15–17] and further into languages like WS-BPEL [13]. In the resulting WS-BPEL process, transitions of the Petri net appear as opaque activities. Refining these activities, one obtains an executable WS-BPEL process. We skip details as they are not necessary for understanding the results in this note.

The synthesized partner provides a communication skeleton for correct interaction with P and is thus valuable beyond witnessing controllability. If one desires to invoke P, he can automatically generate the corresponding code from the description of P. If one does not want to use P arbitrarily, additional constraints may be applied using the techniques of [18].

A particular application of this approach is the automated synthesis of an adapter A between two services P and R. If the composition of P and R is not deadlock-free, an intermediate component may mediate the communication between the two and enforce deadlock freedom. Formally, A is a service such that $P \oplus A \oplus R$ is deadlock-free and can thus be synthesized as a witness for controllability of $P \oplus R$. More precisely, we need to rename interfaces of P and R such that they become disjoint. This way, all communication between P and R will pass A.

4 Semantical constraints

As we synthesize behavior, we are not necessarily interested in the details of the semantics of exchanged messages as such. We are only interested in the impact of semantics on behavioral issues. Experience from adapter synthesis suggests that the main impact of semantical issues is to constrain the ability of a service to manipulate message contents. The semantics determines

Table 1. Examples of semantical constraints in terms of transformation rules

Constraint	Rule	Example pro	Example con
Create a	$\mapsto a$	own password, simple acknowledgement	foreign password
Copy a	$a \mapsto a, a$	address	money transfer, transaction number
Delete a	$a \mapsto$	electronic message	real item (e.g., book)
Transform a into b	$a \mapsto b$ $a \mapsto a, b$	length in feet to length in meters	zipcode to length
Split a into b, c, d	$a \mapsto b, c, d$ $a \mapsto a, b, c, d$	address to name, city, street	the other way round
Merge a, b, c into d	$a, b, c \mapsto d$ $a, b, c \mapsto a, b, c, d$	name, city, street to address	the other way round
Recombine a, b to c, d, e	$a, b \mapsto c, d, e$ $a, b \mapsto a, b, c, d, e$	at reader's discretion	

whether or not the content of a message can be generated, copied, deleted, or computed from the content of other messages.

In [11], we proposed to specify semantical constraints as a set of transformation rules. Each rule consists of two bags saying that the contents of the right hand side messages can be determined from the contents of the left hand side messages, thereby consuming the messages at left hand side. Consumption of involved messages makes sense as real items may be involved while non-consumption of a message may be modeled by re-generating it on the right hand side. For convenience, the universe used for building bags is a set of *semantical entities* that contains but is not restricted to the names of exchanged messages. This way, we have more freedom to model dependencies.

Table 1 lists those semantical constraints which have been proposed in the context of adapter synthesis, together with examples where they make sense as well as examples where they don't make sense. The examples show that the applicability of a rule indeed depends on the semantics of the message contents and cannot be inferred from the service protocol. Consequently, we consider a scenario where the constraint specification is part of the input to the synthesis problem.

There are various ways to generate a specification of semantical constraints. First, they may be generated manually. Since the transformation rules are rather simple, this should not be a problem. Second, they may be inferred using semantic web technology like ontology reasoning. State of the art in this field is beyond the scope of this note. Third, they may become part of the service construction process using some (may be intra-organizational) modeling standard.

5 Synthesis of service behavior in presence of semantical constraints

Our approach (already exercised in [11]) consists of the following steps. Given an open net P and a specification C of semantical constraints, we transform C into an open net S that covers the whole interface of P. S basically manages the message transfer from and to P as well as the transformation of semantical entities according to C. Via a separate interface, it is possible to trigger any activity in S and receive a notification of its execution. In a second step, we synthesize a correctly interacting partner R for $P \oplus S$ using the traditional approach (i.e., not taking care of semantics). Finally, we merge R with S into the final result which can be further optimised using Petri net reduction rules and, if desired, transformed into WS-BPEL.

In this agenda, the construction of S is obviously the crucial part as all other steps rely on existing technology. Consider some given service P and a set C of semantical constraints ranging on a set E of semantical entities. Let I and O be the sets of input and output places of P.

We assume that $I \cap O = \emptyset$ and $I \cup O \subseteq E$. For simplicity of presentation, we assume that for each rule in S, both sides are sets; the general case follows analogously using Petri nets with arc multiplicities.

For defining the service S, we use names from the space $(E \cup C) \times \{e, n, c, r, s\}$, where e, n, c, r, s denote characters instead of variables; hence we assume that these names do not occur in the given service.

The interface of S consists of output places I, input places O (i.e., the interface of P in opposite orientation), and some input and output places specified below. For each entity $e :\ e \in E$, we introduce in service S an internal place (e, c) (c for "copy"). In the initial and final markings, the internal places are empty, although this can easily be generalized in future work.

Service S has three kinds of transitions. For every input place $o :\ o \in O$, there is a transition (o, r) (r for "receive") to move arriving messages from interface place o to their internal place (o, c). For every transformation rule $w :\ w \in C$, there is a transition (w, c) to perform the actual transformation in terms of the internal places. Finally, for every output place $i :\ i \in I$, there is a transition (i, s) (s for "send") to move messages from their internal place (i, c) to interface place i.

Finally, we discuss the additional interface places for the controller. For every input place $o :\ o \in O$, output place (o, n) (n for "notify") notifies an arrived message o. For every transformation rule $w :\ w \in C$, input place (w, e) (e for enable) enables transformation rule w, and output place (w, n) notifies an execution of w. Finally, for every output place $i :\ i \in I$, input place (i, e) enables the delivery of a message i (once available).

Definition 1 (Service *S*). *Let I, O, E, C be as introduced before. The corresponding service S is defined as an open net with the following constituents:*

$$\begin{aligned}
P &= (E \times \{c\}) \ \cup\ P_i \ \cup\ P_o \\
P_i &= O \ \cup\ (C \times \{e\}) \ \cup\ (I \times \{e\}) \\
P_o &= I \ \cup\ (C \times \{n\}) \ \cup\ (O \times \{n\}) \\
T &= (O \times \{r\}) \ \cup\ (C \times \{c\}) \ \cup\ (I \times \{s\}) \\
F &= F_r \cup F_c \cup F_s \\
F_r &= \textstyle\bigcup_{o \in O} \ \{\ [o, (o, r)],\ [(o, r), (o, n)],\ [(o, r), (o, c)]\ \} \\
F_c &= \textstyle\bigcup_{w = X \mapsto Y \in C} \ (\ \{[(m, c), (w, c)] \mid m :\ m \in X\} \ \cup \\
&\qquad \{[(w, e), (w, c)],\ [(w, c), (w, n)]\} \ \cup \{\ [(w, c), (m, c)] \mid m :\ m \in Y\}\ ) \\
F_s &= \textstyle\bigcup_{i \in I} \ \{\ [(i, c), (i, s)],\ [(i, e), (i, s)],\ [(i, s), i]\ \} \\
m_0 &= \underline{0} \\
\Omega &= \{\underline{0}\}
\end{aligned}$$

We use $\underline{0}$ to denote the marking that is zero in every place. By construction, all outputs to P have been obtained from the input of P using the transformation rules only. The actual scheduling of rule applications and message deliveries is left to a controller using the remaining interface.

The inner of S may be unbounded which would complicate further synthesis. For this reason, we pragmatically introduce some capacity on the places of S that, if chosen sufficiently large, should not restrict our results undueley.

Having generated S, it remains to synthesize a partner R of $P \oplus S$ which can be done using the approach of [1, 2]. R basically schedules the application of available actions: it triggers the application of transformations as well as the shipment of messages. Its decisions are based on notifications about incoming messages and applied transformations.

A WS-BPEL process constructed from $R \oplus S$ would contain an opaque activity for each transition, including those that represent the application of transformation rules. In several situations, it is possible to complement the specification of semantical constraints with code snippets that actually implement the specified transformation. In these cases, we may end up with an executable WS-BPEL process that implements the whole interaction with P and is correct by construction.

6 Obeying semantical constraints

In the next section, we wish to establish a result of the following kind: Given a service P and a set C of semantical constraints, if there is any R such that R interacts correctly with P and R obeys the semantical constraints, then $P \oplus S$ is controllable. A result of that kind is only valuable if the definition of "to obey the semantical constraints" is as liberal as possible. In this section, we propose such a definition. For simplicity, we consider only nets where all arc multiplicities are equal to one.

As a starting point, we assume that R has one place for each semantical entity occurring in C. This may be seen as a restriction. Since, however, typical semantical entities are exchanged messages for which there is anyway a representing place, this condition should not be too restrictive. Let P_S be the set of places that represent semantical entities. Let F_S be the set of edges that have their source or sink node in P_S.

The idea of our definition is to mark the application of transformation rules in the normal control flow of an open net. To this end, we use some infinite set U. Each element of U represents the application of a single rule in C. There may be several elements in U that represent the same rule. Elements of U are assigned to those edges which are connected with P_S, i.e. we consider a mapping $\psi : F_S \to U$. This way, access to semantical entities is grouped. u represents a rule $X \mapsto Y$ iff the source places of arcs labeled with u match X, the sink places of arcs labeled u match Y, and each consumption activity causally precedes each production activity. Formally, the first requirements amount to $X = \{p \mid [p,t] \in F_S,\ \psi([p,t]) = u\}$, $Y = \{p \mid [t,p] \in F_S,\ \psi([p,t]) = u\}$. Causal precedence is difficult to formalize as we do not want to rule out open nets with cycles. Therefore, we need to separate different instances of transitions which contribute to a rule. This leads to the second restriction. We require that, for each u, every run of the inner of R can be divided into sequentially arrangeable parts such that each part contains exactly one occurrence of each transition contributing to u (i.e. there is a p such that $\psi([p,t]) = u$ or $\psi([t,p]) = u$). Within each part, we may now require that $\psi([p,t]) = u$ and $\psi([t,p]) = u$ implies that t causally precedes or is equal to t' which formalizes the idea that consumption precedes production.

We say that R obeys C iff a mapping ψ with the discussed properties exists.

It is easy to see that $R \oplus S$ as computed in the previous section obeys C.

7 Completeness

With the definition of the previous section we are now ready to claim completeness of our approach.

Theorem 1. *Consider an open net P and a set C of semantical constraints. If P has a correctly interacting partner that obeys C then $P \oplus S$ is controllable where S is the open net constructed from C as described earlier in this note.*

For proving this theorem, let R be a correctly interacting partner of P that obeys C. We transform R into a correctly interacting partner of $P \oplus S$ using the following ideas.

- Rename input places p of R to (p,n) and output places to (p,e). This way, R talks to S instead of P.
- For each used $u \in U$, introduce new places p_u and q_u. These places control the invocation of rules.

– For each $[p, t]$ with $\psi([p, t]) = u$, an arc $[t, p_u]$; for each $[t, p]$ with $\psi([t, p]) = u$, an arc $[q_u, t]$;
– For each u (where c=$X \mapsto Y$ is the rule represented by u), a transition that consumes $|X|$ tokens from p_u and puts one token on (c, e) as well as a transition that consumes one token from (c, n) and puts $|Y|$ tokens on q_u.

By this construction, a rule is invoked in S after having consumed the corresponding semantical entities in R but before having produced any entity. By our requirements on causal dependencies, the construction does not influence the behavior of R. Thus, the resulting partner interacts correctly with $P \oplus S$.

8 Conclusion

We have shown that the approach of [11] applies to partner synthesis in general. We have further shown that, for a quite liberal definition of "obeying semantical constraints" our approach is complete in the sense that we can synthesize a partner that obeys the constraints iff one exists. This result is, of course, subject to the following shortcomings: First, we are restricted to finite state partners with a given bound on the access of interface places. Second, we have artificially limited the concurrent application of rules and intermediate storage of semantical entities in S to make S finite state as well. Third, technicalities in the definition of "obey C" may be further relaxed. Nevertheless, the completeness result should add confidence into our approach.

References

1. Schmidt, K.: Controllability of open workflow nets. In: Enterprise Modelling and Information Systems Architectures. Volume P-75 of LNI. (2005) 236–249
2. Weinberg, D.: Analyse der Bedienbarkeit. Diplomarbeit, Humboldt-Universität zu Berlin (2004)
3. Moser, S., Martens, A., Häbich, M., Müller, J.: A hybrid approach for generating compatible WS-BPEL partner processes. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 458–464
4. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Proc. ICATPN. Volume 4546 of LNCS. (2007) 321–341
5. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Proc. BPM Workshops. Volume 3812 of LNCS. (2005) 130–143
6. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R., Toumani, F.: Developing adapters for Web services integration. In: Proc. CAiSE. Volume 3520 of LNCS. (2005) 415–429
7. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 65–80
8. Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: Proc. ICSOC. Volume 4294 of LNCS. (2006) 27–39
9. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74**(1) (2005) 45–54
10. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. WWW. (2007) 993–1002
11. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (2008)
12. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Proc. WS-FM. Volume 4937 of LNCS. (2007) 77–91
13. Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into human-readable abstract BPEL processes. In: Proc. Modellierung. Volume P-127 of LNI. (2008) 57–72
14. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidablity of partner existence for open nets. Inf. Process. Lett. (2008) (Accepted for publication).
15. Ehrenfeucht, A., Rozenberg, G.: Partial 2-structures. Acta Informatica **27** (1990) 315–368
16. Desel, J., Reisig, W.: The synthesis problem of Petri nets. Acta Informatica **33** (1996) 297–315
17. Badouel, E., Darondeau, P.: Theory of regions. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS., Springer-Verlag (1996) 529–586
18. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: Business Process Management 2007. Volume 4714 of LNCS. (2007) 271–287

Towards Synthesis of Petri Nets from General Partial Languages

Robert Lorenz

Lehrprofessur für Informatik
Universität Augsburg, Germany
e-mail: robert.lorenz@informatik.uni-augsburg.de

Abstract. In this paper we investigate synthesis of place/transition Petri nets from three different finite representations of infinite partial languages, generalizing previous results.

1 Introduction

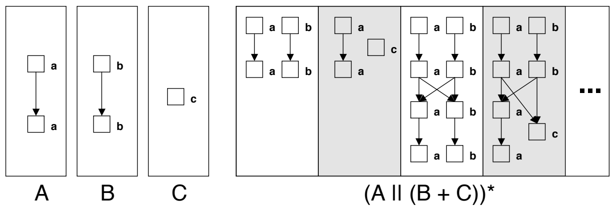


Fig. 1. Partial language given by a term.

In the last two years we generalized the theory of regions for the synthesis of Petri nets from sequential languages and step languages to so called partial languages [LJ06]. A partial language specifies the behaviour of a concurrent system through a possibly infinite set of labeled partial orders (LPOs). Each LPO specifies a run of the system given by a partial order between events labeled by action names. Unordered events are interpreted to be concurrent. The left side of Figure 1 shows three different LPOs, the right side shows a partial language. Through the theory of regions it is possible to compute from a given partial language a Petri net having all specified LPOs as partially ordered runs and having minimal additional behaviour.

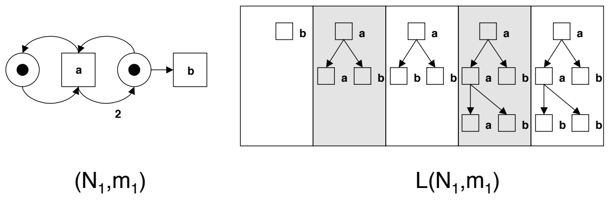


Fig. 2. Partial language without term-based representation.

In this paper we consider classical place/transition Petri nets (p/t-nets). In [LBDM07] we developed an effective synthesis algorithm based on the theory of regions from finite partial languages. In [LBDM08] we generalized this result to such infinite partial languages having a finite term based representation using operators for iteration $(*)$, sequential composition (;), alternative composition $(+)$ and parallel composition $(\|)$. Figure 1 shows some of the LPOs of the infinite partial language given by the term $(A \parallel (B + C))^*$ composing elementary LPOs A, B, C.[1]

Unfortunately only a small class of infinite partial languages can be represented in such a term based form. The Figures 2 and 3 show examples of infinite partial languages which can not be given by a term as above. The main reason for that is, that by the iteration operator it is not possible to append events only to a part of an LPO, but only to the whole LPO. In both cases a p/t-net having the given partial language as its set of (partially ordered) runs is shown.

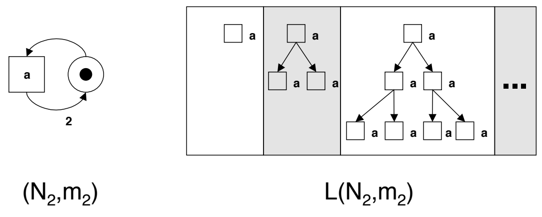


Fig. 3. Partial language without term-based representation.

In this paper we propose three different more general finite representations of infinite partial languages. Each of these representations allows to iteratively append events to parts of LPOs. Therefore, it is possible to represent the finite complete prefix of the branching process of bounded p/t-nets, i.e. we claim that by each of these finite representations the language of (partially ordered runs) of arbitrary bounded p/t-nets can be specified.

Due to lack of space we mostly present the ideas lying behind these finite representations only in an informal way through examples. Finally, very briefly, we suggest how regions could be defined for each of the finite representations.

2 Finite Representations

In this section we introduce three different finite representations of infinite partial languages.

By $\mathbb{N}$ we denote the *nonnegative integers*. $\mathbb{N}^+$ denotes the positive integers. Given a finite set A, the symbol $|A|$ denotes the *cardinality* of A. The set of all *multi-sets* over a set A is the set $\mathbb{N}^A$ of all functions $f : A \to \mathbb{N}$. Given a binary relation $R \subseteq A \times A$, we write aRb to denote $(a, b) \in R$. A *directed graph* is a pair $(V, \to)$, where V is a finite *set of nodes* and $\to \subseteq V \times V$ is called the *set of arcs*. A *partial order* is a directed graph $\mathrm{po} = (V, <)$, where $< \subseteq V \times V$ is irreflexive and transitive.

Definition 1 (Labeled partial order). *A* labeled partial order *(LPO) is a triple* $\mathrm{lpo} = (V, <, l)$*, where* $(V, <)$ *is a partial order and* $l : V \to T$ *is a* labeling function *with* set of labels T.

In our context, a node v of an LPO $(V, <, l)$ is called *event*, representing an occurrence of $l(v)$. Two nodes $v, v' \in V$ are called *independent* if $v \not< v'$ and $v' \not< v$. Notice that by this definition, independence is reflexive. By $\mathrm{co} \subseteq V \times V$ we denote the set of all pairs of independent nodes of V. A *co-set* is a subset $C \subseteq V$ satisfying

[1] Note that for a clearer presentation no transitive edges of the LPOs are drawn.

$\forall x, y \in C : \; x \,\mathrm{co}\, y$. A *cut* is a maximal co-set (w.r.t. set inclusion). For a co-set C of a partial order $(V, <)$ and a node $v \in V \setminus C$ we write $v < C$, if $v < s$ for an element $s \in C$, and $v \,\mathrm{co}\, C$, if $v \,\mathrm{co}\, s$ for all elements $s \in C$. A partial order $(V', <')$ is a *prefix* of a partial order $(V, <)$ if $V' \subseteq V$, $<' = < |_{V' \times V'}$ and $(v' \in V' \wedge v < v') \implies (v \in V')$. Given two partial orders $\mathrm{po}_1 = (V, <_1)$ and $\mathrm{po}_2 = (V, <_2)$, we say that po_2 *is a sequentialization of* po_1 if $<_1 \subseteq <_2$. We use the notations defined for partial orders also for LPOs. If T is the set of labels of $\mathrm{lpo} = (V, <, l)$ then for a set $V' \subseteq V$, we define the multi-set $|V'|_l \subseteq \mathbb{N}^T$ by $|V'|_l(t) = |\{v \in V' \mid l(v) = t\}|$. We consider LPOs only up to isomorphism

Definition 2 (Partial language). *Let T be a set. A set $\mathcal{L}$ of LPOs* $\mathrm{lpo} = (V, <, l)$ *with* $l(V) \subseteq T$ *and* $\bigcup_{(V,<,l) \in \mathcal{L}} l(V) = T$ *is called* partial language over T.

A *net* is a triple (P, T, F), where P is a (possibly infinite) set of *places*, T is a finite set of *transitions* satisfying $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*.

Definition 3 (Place/transition net). *A* place/transition-net *(*p/t-net*)* N *is a quadruple* (P, T, F, W)*, where* (P, T, F) *is a net, and* $W : F \to \mathbb{N}^+$ *is a* weight function.

We extend the weight function W to pairs of net elements $(x, y) \in (P \times T) \cup (T \times P)$ with $(x, y) \notin F$ by $W(x, y) = 0$. A *marking* of a net $N = (P, T, F, W)$ is a function $m : P \to \mathbb{N}$, i.e. a multi-set over P. A *marked p/t-net* is a pair (N, m_0), where N is a p/t-net, and m_0 is a marking of N, called *initial marking*. The occurrence rule of p/t-nets is defined as usual. The non-sequential semantics of a p/t-net can be given by *enabled LPOs*, also called *runs*. An LPO is enabled in a net if the events of the LPO can occur in the net respecting the concurrency relation of the LPO.

Definition 4 (Enabledness). *Let* (N, m_0) *be a marked p/t-net,* $N = (P, T, F, W)$*. An LPO* $\mathrm{lpo} = (V, <, l)$ *with* $l : V \to T$ *is called* enabled w.r.t. (N, m_0) *if for every cut* C *of* lpo *and every* $p \in P$ *there holds* $m_0(p) + \sum_{v \in V \wedge v < C}(W(l(v), p) - W(p, l(v))) \geq \sum_{v \in C} W(p, l(v))$. *Its* occurrence *leads to the marking* m' *given by* $m'(p) = m_0(p) + \sum_{v \in V}(W(l(v), p) - W(p, l(v)))$ *for each* $p \in P$.

The set of of LPOs enabled w.r.t. a given marked p/t-net (N, m_0) *is denoted by* $\mathcal{L}(N, m_0)$. $\mathcal{L}(N, m_0)$ *is called the* partial language of runs *of* (N, m_0).

An alternative characterization of enabled LPOs is through so called process nets. A process net is an acyclic net without conflicts which "unfolds" a p/t-net by representing tokens from some marking of the p/t-net through places (called conditions) and transition occurrences through transitions (called events). Since in a process net the flow relation has no cycles and thus defines a partial order among conditions and events. Omitting the conditions and keeping this partial order between the events yields an enabled LPO, called run underlying the process net. The other way round, each enabled LPO sequentializes the run underlying some process net.

The set of all (alternative) process nets of a p/t-net can be represented by the (possibly infinite) branching process which is an acyclic net including conflicts. In the case the p/t-net is bounded, there is a finite prefix of the branching process (called complete finite prefix) which represents all reachable markings. Roughly speaking, it is determined through cutting the branching process if a marking is repeated. Omitting the conditions and keeping the partial order and conflict relation between the events yields a so called prime event structure (underlying the finite complete prefix) which represents a set if runs underlying process nets.

Note that the partial language of runs of a p/t-net is always prefix- and sequentialization-closed. In examples and Figures we often do not draw all prefixes and sequentializations but assume that they are present.

2.1 Identification of states

The finite complete prefix (resp. its underlying prime event structure) of a bounded p/t-net can be represented on the level of languages by a finite set of LPOs. Of course, from this finite set the complete non-sequential behavior can only be re-constructed, if one keeps the information, at which points the branching process was cut w.r.t. which repeated marking. This can be done by remembering, which prefixes of which LPO lead to the same marking. That means, a possibility for specifying the non-sequential behavior of bounded p/t-nets is through a finite set of LPOs together with some equivalence relation on prefixes of these LPOs.

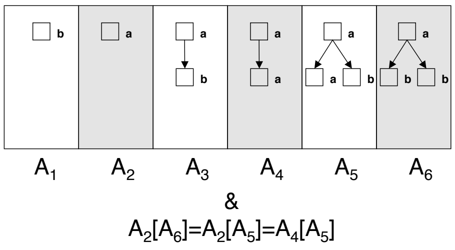


Fig. 4. Set of LPOs with identification of states representing $L(N_1, m_1)$.

If two prefixes are equivalent, this means that all events occurring after the one prefix also can occur after the second prefix and vice versa. Infinite behavior is specified for example if a prefix is prefix of an equivalent prefix. Figure 4 shows, how by this method the language $L(N_1, m_1)$ from Figure 2 can be given. The equation $A_2[A_6] = A_2[A_5] = A_4[A_5]$ means that after occurrence of A_2 in A_6 the same marking is reached as after occurrence of A_2 in A_5 or after occurrence of A_4 in A_5. Therefore, after the occurrence of A_4, the same events as after A_2 in A_5 or A_6 can occur and so on. Also $L(N_2, m_2)$ from Figure 3 can be represented this way (see Figure 5). This means, that through identifying states also the non-sequential behavior of unbounded nets can be specified (at least in some cases).

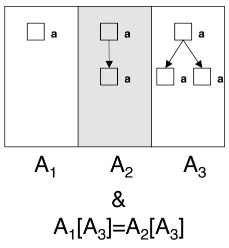


Fig. 5. Set of LPOs with identification of states representing $L(N_2, m_2)$.

2.2 Partial Iteration

In [LBDM08] we introduced a term-based representation of infinite partial language. These terms, called *composed runs*, are build through composing inductively (elementary) LPOs from some given finite set of LPOs $\mathcal{A}$. LPOs can be composed sequentially (;), alternatively ($+$) and parallel ($\|$) and can be iterated ($*$). That means each LPO $A \in \mathcal{A}$ is a composed run and if α, β are composed runs, then also $\alpha;\beta$, $\alpha + \beta$, $\alpha \parallel \beta$ and α^* are composed runs. Each composed run represents a set of LPOs, where an elementary LPO A represents the one-LPO set $L(A) = \{A\}$. The composed run $\alpha;\beta$ represents the set $L(\alpha;\beta) = \{A;B \mid A \in \alpha,\ B \in \beta\}$, $\alpha + \beta$ the set $L(\alpha+\beta) = L(\alpha) \cup L(\beta)$, $\alpha \parallel \beta$ the set $L(\alpha \parallel \beta) = \{A \parallel B \mid A \in \alpha,\ B \in \beta\}$ and α^* the set $L(\alpha^*) = \{A_1; ...; A_n \mid A_i \in \alpha\}$. On the level of LPOs $A;B$ means that each event in A precedes each event in B and $A \parallel B$ means that there is no order between events in A and in B.

Such a representation of partial languages by composed runs is quite restrictive as shown in the introduction, because through sequential composition and iteration it is not possible to append an LPO only to parts of some previous LPO. We therefore introduce here the possibility to iterate and sequentially compose LPO w.r.t. an "interface" specifying to which parts of a previous LPO a subsequent LPO is appended. Such an interface is given through an LPO I connecting events in the previous LPO to minimal events in the subsequent LPO. The composition w.r.t. to such an interface I is denoted by $*_I$ resp. $;_I$ and is realized w.r.t. the ordering given by I.

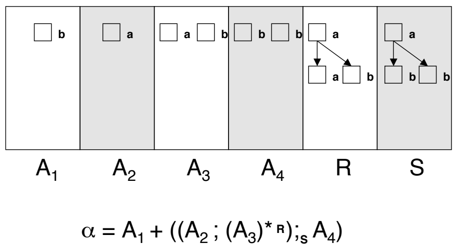


Fig. 6. Composed run with partial iteration representing $L(N_1, m_1)$.

Figure 6 shows, how by this method the language $L(N_1, m_1)$ from Figure 2 can be specified: The LPO A_3 is iterated through appending it only to the a-labeled event and finally A_4 also is appended only to the a-labeled event. Note that it is in principle also possible to represent $L(N_2, m_2)$ through $A_1; (A_1 \parallel A_1)*_{A_1;(A_1 \| A_1)}$. But the interpretation of this expression is not totally clear because there are two possibilities to use the interface $A_1; (A_1 \parallel A_1)$ to iterate $A_1 \parallel A_1$. One interpretation is that only one of the possibilities can be applied, another is that both possibilities can be applied in parallel (and only in this second case $L(N_2, m_2)$ is represented).

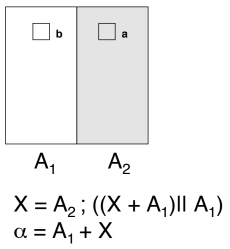


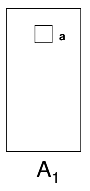
Fig. 7. Composed term with recursion representing $L(N_1, m_1)$.

2.3 Recursion

Another possibility to generalize composed runs is to equip them with recursion. Through recursion it is possible specify that some behavior is repeated at certain points of a composed run. For this also variables can be used in a composed run. Each variable represents a set of LPOs. The set of LPOs specified through a variable X is given through an equation $X = \alpha(X)$, where $\alpha(X)$ is a composed run including X (X need not be minimal in $\alpha(X)$). The interpretation of such an equation is, that each occurrence of X on the right side may be replaced by the empty LPO or by $\alpha(X)$ and so on. It is in general also possible that there are more variables in one composed run and that there are more equations.

Figure 7 shows, how by this method the language $L(N_1, m_1)$ from Figure 2 can be given. Figure 8 shows, how by this method the language $L(N_2, m_2)$ from Figure 3 can be given.

3 Synthesis



$X = A_1 ; (X \parallel X)$
$\alpha = X$

Fig. 8. Composed term with recursion representing $L(N_2, m_2)$.

The general ideas of region based synthesis of p/t-nets from partial languages $\mathcal{L}$ are as follows: The set of transitions of the synthesized net is the finite set of labels of $\mathcal{L}$. Places are defined by their initial marking and the weights on the arcs connecting them to transitions. Two kinds of places can be distinguished. In the case that there is an LPO specified in $\mathcal{L}$ which is no run of the net which has only the one considered place, this place restricts the behaviour too much. Such places are *non-feasible (w.r.t. $\mathcal{L}$)*. In the other case, the considered place is *feasible (w.r.t. $\mathcal{L}$)*. The aim is to add enough feasible places in order exactly reproduce the specified behavior.

Feasible places are computed through so called token flow regions which are defined on the level of the partial language [LJ06]: If two events x and y satisfy $x < y$ in an LPO $\mathrm{lpo} = (V, <, l) \in \mathcal{L}$, this specifies that the corresponding transitions $l(x)$ and $l(y)$ may be causally dependent. Such a causal dependency arises exactly if the occurrence of the transition $l(x)$ produces one or more tokens in a place, and some of these tokens are consumed by the occurrence of the other transition $l(y)$. Such a place can be defined as follows: Assign to every edge (x, y) of an LPO in $\mathcal{L}$ a natural number $r(x, y)$ representing *the number of tokens which are produced by the occurrence of $l(x)$ and consumed by the occurrence of $l(y)$ in the place to be defined*. For this, we extend each LPO $\mathrm{lpo} \in L$ by an initial event v_{lpo} and a final event, representing transitions producing the initial marking and consuming the final marking (after the occurrence of lpo). A feasible place p_r is then defined by assigning for each extended LPO $\mathrm{lpo} = (V, <, l) \in L$ a natural number $r(x, y)$ to each edge (x, y) function r, where it holds that (IN): $In(y, r) = \sum_{x <^{\star} y} r(x, y) = \sum_{x <^{\star} z} r(x, z) = In(z, r)$ for $l(y) = l(z)$, (OUT): $Out(y, r) = \sum_{y <^{\star} x} r(y, x) = \sum_{z <^{\star} x} r(z, x) = Out(z, r)$ for $l(y) = l(z)$ and $(INIT)$: $Out(v_{\mathrm{lpo}_1}, r) = Out(v_{\mathrm{lpo}_2}, r)$ for $\mathrm{lpo}_1, \mathrm{lpo}_2 \in \mathcal{L}$. We call $In(y, r)$ the *intoken flow* of y which is interpreted as the weight of the arc connecting the new place p_r with the transition $l(y)$ (i.e. $W(p_r, l(y)) = In(y, r)$). We call $Out(y, r)$ the *outtoken flow* of x, which is interpreted as the weight of the arc connecting the transition $l(x)$ with the new place p_r (i.e. $W(l(y), p_r) = Out(y, r)$). The outtoken flow of v_{lpo} is called *initial flow* and is interpreted as the initial marking of the new place p_r (i.e. $m_0(p_r) = Out(z, r)$). The value $r(x, y)$ is called the *token flow* between x and y.

A function r satisfying (IN), (OUT) and $(INIT)$ is called *region*. The main result of [LJ06] is that the set of places corresponding to regions of a partial language equals the set of feasible places w.r.t. this partial language.

This notion of regions can easily be adapted to each of the proposed finite representations. Namely, in each case a token flow function r need to fulfil requirements additional to (IN), (OUT) and $(INIT)$. In case, a partial language is given by a finite set of LPOs and an equivalence relation on prefixes of those LPOs, we require that

- r satisfies (IN), (OUT) and $(INIT)$ on the finite set of LPOs.
- r satisfies that for equivalent prefixes the sum of token flows on edges leaving one prefix equals the sum of token flows on edges leaving the other prefix.

In case, a partial language is given by a composed run using partial iteration, we require the same properties as for composed runs introduced in [LBDM08]. There an additional requirement was introduced for the so called set of iterated LPOs postulating that the initial and the final token flow of such iterated LPOs should be equal. The only difference now is that the initial and final token flow of such LPOs is computed in another way, namely w.r.t. the given interface. In case, a partial language is given by a composed run equipped with recursion equations, we require

- the same as for composed runs and additionally that
- for each equation $X = \alpha(X)$ the intial flow of $\alpha(X)$ equals the sum of token flows on edges ingoing an occurrence of X in $\alpha(X)$ for each such occurrence.

All these additional requirements can be represented as homogenous linear inequations as it is the case for (IN), (OUT) and $(INIT)$. Thus effective solution algorithms can be adapted.

References

[LBDM07] LORENZ, R. ; BERGENTHUM, R. ; DESEL, J. ; MAUSER, S.: Synthesis of Petri Nets from Finite Partial Languages. In: *ACSD*, IEEE Computer Society, 2007, S. 157–166

[LBDM08] LORENZ, R. ; BERGENTHUM, R. ; DESEL, J. ; MAUSER, S.: Synthesis of Petri Nets from Infinite Partial Languages. In: *Proceedings of ACSD*, 2008, S. 170 – 179

[LJ06] LORENZ, R. ; JUHÁS, G.: Towards Synthesis of Petri Nets from Scenarios. In: DONATELLI, S. (Hrsg.) ; THIAGARAJAN, P. S. (Hrsg.): *ICATPN* Bd. 4024, Springer, 2006 (Lecture Notes in Computer Science), S. 302–321

[LMB07] LORENZ, R. ; MAUSER, S. ; BERGENTHUM, R.: Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios. In: KLEIJN, J. (Hrsg.) ; YAKOVLEV, A. (Hrsg.): *ICATPN* Bd. 4546, Springer, 2007 (Lecture Notes in Computer Science), S. 342–361

Decompositional Computation of Operating Guidelines Using Free Choice Conflicts

Niels Lohmann[*]

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
niels.lohmann@uni-rostock.de

Abstract. An operating guideline (OG) for a service S finitely characterizes the (possibly infinite) set of all services that can interact with S without deadlocks. This paper presents a decompositional approach to calculate an OG for a service whose underlying structure is acyclic and contains free-choice conflicts. This divide-and-conquer approach promises to be more efficient than the classical OG computation algorithm.

1 Introduction

In the paradigm of service-oriented computing, a *service* is a component that offers a functionality over a well-defined interface and is discoverable and accessible via a unique identifier. By composing several services, complex tasks (e. g., inter-organizational business processes) can be realized. Thereby, the correct interplay of distributed services is crucial to achieve a common goal.

Recent literature [1] proposed an *operating guideline* (OG) of a service S as finite characterization of all (partner) services that communicate correctly (i. e., without deadlocks or livelocks) with S. Applications of OGs include the realization the "find" and "publish" operations of service brokering, as well as the analysis, construction, and correction of services. Unfortunately, the algorithm to calculate an OG for a service has exponential complexity in both the service's state space and the size of the interface. In this paper, we propose a decompositional divide-and-conquer approach to calculate OGs for service models that contain free choice conflicts.

In Sect. 2, we recall some necessary definitions. Section 3 introduces decomposition of service models and describes how OGs can be calculated decompositionally. In Sect. 4, we analyze which constructs of industrial specification languages meet the requirements of the decomposition. Section 5 concludes the paper and discusses future work.

2 Background

We use *open nets* [2] to model services. Open nets extend classical Petri nets [3] with an interface $I = (P_{in} \cup P_{out}) \subseteq P$ to explicitly model asynchronous message exchange and a set of final markings Ω modeling desired final states of the service. Two open nets N and M can be *composed* (denoted by $N \oplus M$) by merging their interfaces accordingly (N's input places with M's output places, and vice versa). Thereby, the *inner structures* of N and M (i.e., the open net without interface)

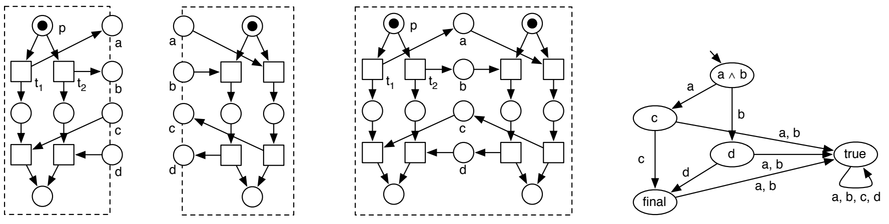
[*] funded by the DFG project "Operating Guidelines for Services" (WO 1466/8-1)

are assumed to be disjoint. An open net is *acyclic* if the reachability graph of its inner structure is acyclic. An open net *weakly terminates* if, from every reachable marking, a final marking is reachable.

Definition 1 (Controllability, strategy). *Let N be an open net. N is* controllable*, iff there exists an open net M such that $N \oplus M$ is weakly terminating. Then M is called a* strategy *for N. Denote the set of all strategies for N by $\mathit{Strat}(N)$.*

In [1], the concept of an *operating guideline* (OG) was introduced. The operating guideline OG_N for a service N is a finite automaton whose states are annotated with Boolean formulae. It characterizes a (possibly infinite) set of services, denoted by $\mathit{Comply}(OG_N)$. In fact, it *exactly* characterizes the set of strategies of N.

Theorem 1 ([1]). *Let OG_N be an operating guideline for an open net N. Then $\mathit{Comply}(OG_N) = \mathit{Strat}(N)$.*



(a) N (b) M (c) $\mathsf{N} \oplus \mathsf{M}$ (d) OG_N

Fig. 1. Open net N, strategy M, composition $\mathsf{N} \oplus \mathsf{M}$, and operating guideline OG_N

Example Figure 1 depicts an open net N. The net is controllable, as there exists a strategy M which first receives either an a or a b message and then responds with a c or a d message, resp. This and all other strategies are characterized by the operating guideline OG_N. The conjunction $\mathsf{a} \wedge \mathsf{b}$ annotated to the initial node states that a strategy must be ready to initially both receive an a message *and* a b message. The node with the true formula is a technical necessity. Though it will never be reached in a composition (e. g., after having received an a message, the further receipt of either a or b is impossible, because N will not send these messages), such respective branches may be still part of a strategy, because they do not jeopardize weak termination.

3 Decomposition

In a Petri net, a place with more than one transition in its postset models a conflict.

Definition 2 (Conflict cluster, free choice). *Let $x \in P \cup T$ be a node of a net. The* conflict cluster *x, denoted by $[x]$, is the minimal set of nodes such that:*

- $x \in [x]$.

– *If $p \in [x]$ for a place $p \in P$, then $p^\bullet \subseteq [x]$.*
– *If $t \in [x]$ for a transition $t \in T$, then ${}^\bullet t \subseteq [x]$.*

$[x]$ is free choice *if for all $t, t' \in [x] \cap T$ holds: either ${}^\bullet t \cap {}^\bullet t' = \emptyset$ or ${}^\bullet t = {}^\bullet t'$.*

Given a marking of a net, this marking either enables all transitions in a free choice conflict cluster or none of them. We exploit this property by using the transitions of a free choice conflict cluster to decompose the net. For each possible outcome of the conflict, we define one net in which only this transition is present and all others are removed together with their adjacent arcs.

Definition 3 (Decomposition). *Let $N = [P, T, F, m_0, \Omega]$ be an open net and C a free choice conflict cluster of N with $C \cap T = \{t_1, \ldots, t_m\}$. The* decomposition *of N w.r.t. C is the set $\{N_1, \ldots, N_m\}$ with $N_i = [P, T \backslash \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, F \setminus ((P \times \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}) \cup (\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\} \times P)), m_0, \Omega]$, for $i \in \{1, \ldots, m\}$.*

Theorem 2. *Let N be a safe acyclic open net and $\{N_1, \ldots, N_m\}$ be its decomposition w.r.t. a free-choice conflict cluster C of N with $C \cap T = \{t_1, \ldots, t_m\}$. Then $Strat(N) = \bigcap_{i=1}^{m} Strat(N_i)$.*

Proof. We prove the equality by showing mutual set inclusion.

$\subseteq$: Let $M \in Strat(N)$. We will show by contradiction that $M \in \bigcap_{i=1}^{m} Strat(N_i)$. Assume $M \notin \bigcap_{i=1}^{m} Strat(N_i)$. Then $M \oplus N_i$ contains a deadlock for a net N_i. Let $m_0 \xrightarrow{\sigma} m_d$ be a transition sequence to this deadlock in $M \oplus N_i$. This sequence is also realizable in $M \oplus N$. There, m_d might activate a transition not present in $M \oplus N_i$, which can only be a transition in $(C \cap T) \setminus \{t_i\}$. As C is a free choice conflict cluster, m_d also activates t_i in $M \oplus N_i$ which contradicts the assumption that $M \oplus N_i$ contains a deadlock. Consequently, $M \oplus N_i$ is deadlock free and $M \in Strat(N_i)$. Repeating the arguments, we can conclude $M \in \bigcap_{i=1}^{m} Strat(N_i)$.

$\supseteq$: Let $M \in \bigcap_{i=1}^{m} Strat(N_i)$. We will show by contradiction that $M \in Strat(N)$. Assume $M \notin Strat(N)$. Then $M \oplus N$ contains a deadlock. Let $m_0 \xrightarrow{\sigma} m_d$ be a transition sequence to this deadlock in $M \oplus N$. There are two cases:
  – σ contains a transition of $C \cap T$. Then, by definition of the decomposition, there exists a net N_i such that σ is realizable in $M \oplus N_i$, because due to safeness and acyclicity, transitions in $C \cap T$ can occur at most once in σ.
  – σ contains no transition of C. Then σ is realizable in $M \oplus N_i$, for any $1 \leq i \leq n$.

Both cases would contradict the assumption that $M \in \bigcap_{i=1}^{m} Strat(N_i)$. Hence, $M \oplus N$ is deadlock free and $M \in Strat(N)$.

□

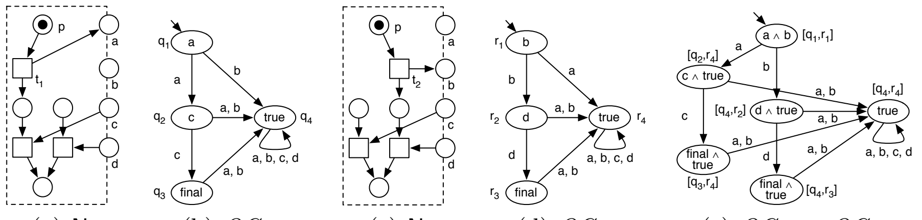
Theorem 1 describes the relationship between the strategy set of a service and its OG. The intersection of strategy sets can be related to OGs using the *product operator* [4]. The product of two operating guidelines OG_N and OG_M, denoted by $OG_N \otimes OG_M$, is constructed similar to the product automaton for classical finite automata. In addition, the formula annotated to a state $[q_1, q_2]$ of the product is defined to be the conjunction of the formula annotated to q_1 and that of q_2.

Theorem 3 ([4]). *Let OG_N, OG_M be operating guidelines. Then $Comply(OG_N \otimes OG_M) = Comply(OG_N) \cap Comply(OG_M)$.*

This result allows us to express Theorem 2 in terms of operating guidelines:

Corollary 1. *Let N be a safe acyclic open net and $\{N_1, \ldots, N_m\}$ be its decomposition w.r.t. a free-choice conflict cluster C of N with $C \cap T = \{t_1, \ldots, t_m\}$. Then $Comply(OG_N) = Comply(OG_{N_1} \otimes \cdots \otimes OG_{N_m})$.*

We are now able to calculate an operating guideline for N by calculating the operating guidelines for the decomposition of N, followed by calculating the product of the operating guidelines. Note that Theorem 2 does not require the whole net to be free choice, but only the conflict cluster under consideration.



(a) N_1 (b) OG_{N_1} (c) N_2 (d) OG_{N_2} (e) $OG_{\mathsf{N}_1} \otimes OG_{\mathsf{N}_2}$

Fig. 2. Decomposed open nets N_1 and N_2 with operating guidelines OG_{N_1} and OG_{N_2} and the product operating guideline $OG_{\mathsf{N}_1} \otimes OG_{\mathsf{N}_2}$

Example (cont.) The net N in Fig. 1 contains a free choice conflict cluster $\{\mathsf{p}, \mathsf{t}_1, \mathsf{t}_2\}$ and can be decomposed into the nets N_1 and N_2, depicted in Fig. 2. The respective OGs characterize the strategies for the decomposed nets. To determine the intersection of these strategy sets, the product $OG_{\mathsf{N}_1} \otimes OG_{\mathsf{N}_2}$ needs to be constructed. As described earlier, it is the product of the underlying automata, and each state is annotated with the conjunction of the respective formulae. For example, the annotation of state $[\mathsf{q}_1, \mathsf{r}_1]$ is the conjunction of the formulae of q_1 (a) and r_1 (b). The resulting product $OG_{\mathsf{N}_1} \otimes OG_{\mathsf{N}_2}$ is equivalent to OG_{N} (cf. Fig. 1); that is, it characterizes the same set of strategies.

The advantage of the decompositional OG calculation is the reduced complexity of the intermediate results. Though the OGs of the decomposed nets might have more nodes, the state space of the decomposed nets is usually much smaller. Furthermore, the product operator's associativity allows to interleave the OG calculation and the product construction.

4 Applications

Though the requirements of Theorem 2 (safeness, acyclicity) are very restrictive, the decompositional approach to calculate an OG can still be used for industrial specification languages. In the following, we evaluate which features of the languages BPEL [5], BPMN [6], and UML2 [7] activity diagrams may be used to while still meeting the requirements of Theorem 2.

BPEL For BPEL there exists a feature-complete Petri net semantics [8] which allows to translate a BPEL process into safe open nets. The net is cyclic only if activities for repetitive execution (`while`, `repeatUntil`, and sequential `forEach`) or event handlers are used in the process. Several patterns contain conflicts of which many are not free-choice. However, the following are:

– decisions modeled with the `if` activity (in case XPath errors are not modeled),
– transition conditions to set control links within a `flow` activity, and
– leaving the process's positive control flow (throwing a the first fault).

Furthermore, conflicts can depend on each other. For example, whether or not to skip an activity during dead path elimination is a non-free choice, yet dependent on the setting of the respective control links, which in turn is a free choice decision. Hence, when decomposing the net using such a "dominant" conflict, several "dependent" decisions become deterministic.

BPMN Dijkman et al. [9] defined a Petri net semantics for a subset of BPMN. The resulting Petri net is safe if the control flow does not contain a *lack of synchronization*. This situation can arise if gateways are not nested properly (e.g., the control flow splits using an AND-gateway, but joins using an XOR-gateway). Such models contain obvious design flaws.

The nets are acyclic if the control flow is acyclic and no activities with explicit loop annotation are used. Again, many occurring conflicts are not free-choice, especially when exception flow is modeled. However, decision gateways can be translated into free choice conflicts.

UML-AD UML2 activity diagrams have a very close relationship to BPMN and Petri nets. An activity diagram can be translated into an acyclic Petri net if its control flow is acyclic. In case the diagram contains no lack of synchronization, the translation results a safe Petri net. Additionally, the whole net (i. e., every conflict cluster) is free choice if no pinsets are used.

Table 1. Language constructs for acyclic safe Petri nets and free choice conflicts.

language	acyclic Petri net	safe Petri net	free choice conflicts
BPEL	✘ `while`	✔ always safe	✔ `if` branches
	✘ `repeatUntil`		✔ transition conditions
	✘ sequential `forEach`		✔ throwing first fault
	✘ event handlers		✘ `pick`
BPMN	✘ loop activities	✘ lack of synchronization	✔ data-based gateways
	✔ acyclic control flow		✔ inclusive gateways
			✔ timeout event gateway
UML-AD	✔ acyclic control flow	✘ lack of synchronization	✔ all, if no pinsets are used

Table 1 summarizes the language constructs that are forbidden or that guarantee acyclic and safe Petri nets, and that yield free choice conflicts. The latter constructs can be used to discover free choice conflict clusters already during the translation of a process described in BPEL, BPMN or UML-AD into Petri nets. This allows for avoiding an a-posteriori discovery of free choice conflict clusters.

We implemented the described decomposition approach in the compiler BPEL2oWFN [8] which translates a BPEL process into a set of decomposed open nets. For these nets, the OGs can be calculated using the tool Fiona [10], which also implements the calculation of product operating guidelines.[1]

5 Conclusion

We presented a decompositional approach that uses free-choice conflict clusters to decompose a safe acyclic open net. The operating guidelines for the resulting nets can be calculated independently and subsequently merged using the product operator. Hence, the calculation can be seen as a divide-and-conquer approach to calculate operating guidelines.

Both the calculation of the OGs for the decomposed nets and the product operators are currently implemented to cope with arbitrary nets and OGs, resp. In future work we plan to adjust these algorithms to exploit the simpler structure of the intermediate constructs. In particular, we plan to study free choice open net, because they can be decomposed into conflict free open nets for which the OG construction should be less complex.

In addition, the requirements of Theorem 2 might be relaxed. For example, the theorem still holds if every transition sequence marks the conflict cluster at most once. This requirement can also be fulfilled by open nets which are cyclic or non-safe.

References

1. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341
2. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. AMCT **1**(3) (2005) 35–43
3. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
4. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007. Volume 4714 of LNCS., Springer (2007) 271–287
5. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. OASIS Standard, 11 April 2007, OASIS (2007)
6. OMG: Business Process Modeling Notation (BPMN) Version 1.0. OMG Final Adopted Specification, OMG (2006)
7. OMG: Unified Modeling Language (UML). Version 2.1.2, OMG (2007)
8. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007. Volume 4937 of LNCS., Springer (2008) 77–91
9. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology (2008) (Accepted for publication).
10. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006. Volume 4102 of LNCS., Springer (2006) 17–32

[1] Both tools are available for download at `http://service-technology.org/tools`.

An Approach to Tackle Livelock-freedom in SOA

Christian Stahl[1,2,⋆] and Karsten Wolf[3,⋆⋆]

[1] Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
stahl@informatik.hu-berlin.de

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[3] Universität Rostock, Institut für Informatik
18051 Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. We calculate a fixed finite set of *state space fragments* for a service P, where each fragment carries a part of the whole behavior of P. By composing these fragments according to the behavior of a service R we build the state space of their composition $P \oplus R$ which can be checked for deadlocks and livelocks. We show that this approach is applicable to realize a "find" request by a service R with a provided service P in SOA.

1 Introduction

In the paradigm of service-oriented computing (SOC) a *service* serves as a building block for designing flexible business processes by composing multiple services. Service-oriented architectures (SOA) serve as an enabler for publishing services via the Internet such that these services can be automatically found. By dynamically binding published services with other services, a composed service that achieves certain business goals can be designed.

In SOA, we would like to answer a "find" request by a service R with a provided service P such that $P \oplus R$ forms a sound, i.e. a deadlock-free and livelock-free system. The apparent approach is to have P (or a public view of P) stored in the repository and to construct $P \oplus R$ for checking the absence of deadlocks and livelocks upon "find". This approach is, however, not feasible due to state space explosion, and the necessity to have P (or a formally equivalent public view) stored in the repository. State space reduction during the construction of $P \oplus R$ might help a lot but needs to be performed for each "find" request.

Another approach is to publish an operating guideline [1] for each service P; that is, an operational description of all services R such that $P \oplus R$ is sound. To answer a "find" request by R, one has to check whether R matches with the operating guidelines of P [1]. However, in this approach soundness is restricted to deadlock-freedom so far and hence livelocks in $P \oplus R$ are possible.

⋆ Funded by the DFG project "Substitutability of Services" (RE 834/16-1).
⋆⋆ Supported by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

In this paper, we propose a novel approach. We still build the state space $P \oplus R$, but not from operational descriptions of P and R. Instead, we calculate a finite set of *state space fragments* for a service P. Each fragment carries a *part* of the whole behavior of P. These fragments are published in a repository. Upon a "find" request by a service R, the state space of $P \oplus R$ is calculated by composing fragments of P according to the behavior of R. The resulting state space can then be checked for deadlocks and livelocks using a model checker. The approach has two advantages:

1. The construction of fragments and their internal state space reduction is done once for each published service at the "publish" phase. That way, computational efforts are shifted from "find" to "publish". This is a clear advantage as we expect the number of "find" to be much higher than the number of "publish".
2. When reducing the size of fragments, we can apply reduction techniques which are different from standard state space reduction techniques used in model checking. In fact, we may reduce the transition system *after* having computed it.

Section 2 formalizes fragments, shows how fragments for a given service P can be computed, and it presents how the state space $P \oplus R$ can be built from the fragments of P. Section 3 sketches several abstractions to condense fragments while preserving deadlocks and livelocks and, finally, Sect. 4 concludes the paper.

2 Calculating State Spaces From Fragments

2.1 Formalizing Fragments

In this section, we define fragments and connections between these fragments.

A *(state space) fragment* $\mathit{Frag} = (V, E, F)$ is a graph that consists of a set V of *nodes*, a set $E \subseteq V \times V$ of (directed) *edges*, and a set $F \subseteq V$ of *final nodes*. We assume that different fragments have disjoint sets of nodes.

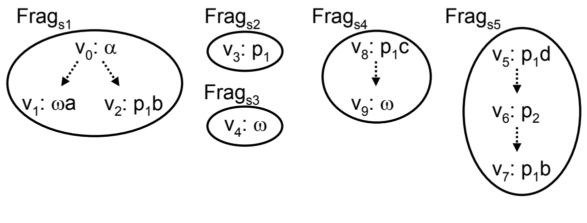
Let x be an element of some fixed set M. An *instance* $\mathit{Frag}(x)$ of a fragment Frag is built by renaming the constituents as follows: $v \mapsto [v, x]$, $e = [v_1, v_2] \mapsto [[v_1, x], [v_2, x]]$ for all $v \in V$ and $e \in E$. That way, the structure is preserved but the nodes get previously unused names.

To plug different fragments yielding again a state space, we define connections which link states of one fragment to states of another fragment. A *connection* $C_{\mathit{Frag}_1, \mathit{Frag}_2}$ between fragments Frag_1 and Frag_2 is a subset of $V_{\mathit{Frag}_1} \times V_{\mathit{Frag}_2}$.

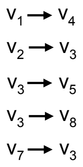
If $C_{\mathit{Frag}_1, \mathit{Frag}_2}$ is a connection between fragments Frag_1 and Frag_2, then $C_{\mathit{Frag}_1, \mathit{Frag}_2}(x, y) = \{([v_1, x], [v_2, y]) \mid (v_1, v_2) \in C\}$ is a connection between $\mathit{Frag}_1(x)$ and $\mathit{Frag}_2(y)$.

Consider the fragments and the connections depicted in Fig. 1. For instance, we have fragment $\mathsf{Frag}_{\mathsf{s1}} = (\{\mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2\}, \{(\mathsf{v}_0, \mathsf{v}_1), (\mathsf{v}_0, \mathsf{v}_2), \emptyset\})$ and connection $C_{\mathsf{Frag}_{\mathsf{s1}}, \mathsf{Frag}_{\mathsf{s3}}} = \{(\mathsf{v}_1, \mathsf{v}_4)\}$. Thereby, v_0 relabels α, v_1 relabels $\omega\mathsf{a}$, etc.

Given a set of fragments $\mathit{Frag}_1, \ldots, \mathit{Frag}_n$ and connections $C_1, \ldots, C_m$, a transitions system $\mathit{TS} = (V, E)$ is defined by $V = \bigcup_{k=1}^{n} V_{\mathit{Frag}_k}$ and $E =$



(a) Fragments



(b) Connections

Fig. 1. Fragments and connections. Dotted (solid) lines denote fragment internal transitions (connections).

$\bigcup_{i=1}^{n} E_{Frag_i} \cup \bigcup_{j=1}^{m} E_{C_j}$. Thereby, several instances of one and the same fragment or connection may be used to build TS.

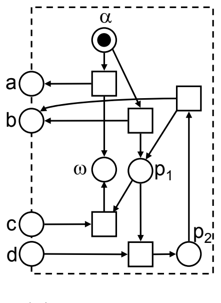
In the following, we introduce our service model open nets and show how fragments and connections of an open net can be calculated.

2.2 Open Nets and Most Permissive Strategy

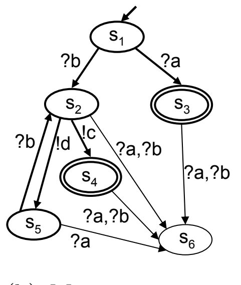
We use *open nets* as a service model. An open net N consists of a Petri net together with an *interface*. The interface is divided into a set of *input places* and *output places*. Input places have an empty preset, output places have an empty postset. Furthermore, N has a distinguished *initial marking* m_0, and a set Ω of *final markings* such that no transition of N is enabled at any $m \in \Omega$. We further require that in the initial and the final markings the interface places are not marked.

The behavior of an open net is defined using the standard Petri net semantics [2]. With $R_N(m_0)$ we denote the set of reachable markings of N.

For example, the open net P depicted in Fig. 2(a) has an initial marking $m_{0P} = [\alpha]$ and the set of final markings is defined by $\Omega_P = \{[\omega]\}$. P has two input places c and d and two output places a and b that are depicted on the dashed frame.



(a) Open net P



(b) Most permissive strategy R^* for P

Fig. 2. Open net P and its most permissive strategy R^*. $?x$ ($!x$) denotes a sending (receiving) message x that produces a token on input place x (consumes a token from output place x) in P.

As a correctness criterion for an open net N we require the absence of deadlocks and livelocks in N. N is *deadlock-free and livelock-free* if for all reachable markings $m \in R(m_0)$, $R_N(m) \cap \Omega_N \neq \emptyset$.

For the *composition* of two open nets M and N, we require that the input places of M are the output places of N and vice versa. M and N can be composed by merging input places of M with equally labeled output places of N and vice versa.

As we are interested in composing open nets such that the composition is deadlock-free and livelock-free we define the notion of a strategy. An open net M is a *strategy* for an open net N if $M \oplus N$ is deadlock-free and livelock-free.

In [1] it has been proven that there always exists a *most permissive strategy* R^* for an open net N that has richer behavior than every other strategy for N.

The most permissive strategy for P (of Fig. 2(a)) is depicted in Fig. 2(b). It is an automaton (which can easily be transformed in a state machine and by adding an input (output) place for each $?x$ $(!x)$ to an open net) with initial state $\mathsf{s_1}$ and two final states $\mathsf{s_3}$ and $\mathsf{s_4}$. State $\mathsf{s_6}$ is depicted for technical purposes only. Every edge to $\mathsf{s_6}$ shows a possible set of messages R^* can receive but that will never occur because P cannot send them.

For R^* we can prove the following useful property.

Lemma 1. *If R is a strategy of P, then R^* weakly simulates R.*

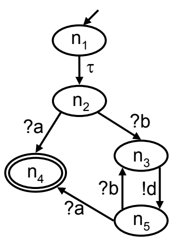
The converse does not hold in general. The automaton R^* (see Fig. 2(b)) weakly simulates R (see Fig. 3(a)) but $P \oplus R$ has a livelock (as we will see later on). In fact, R is an example why the operating guideline approach in [1] is not applicable to tackle livelock-freedom.

When computing R^* we have the information needed to calculate the fragments and connections of P. Each state s of R^* is a fragment. In each state s, R^* has knowledge about the possible markings of P in s. These markings (together with their transitions) are the nodes and the edges of the fragment. Figure 1 shows the fragments and the connections of P. $\mathsf{Frag_{s1}}$ is the fragment derived from $\mathsf{s_1}$, $\mathsf{Frag_{s2}}$ from $\mathsf{s_2}$ and so on. For $\mathsf{s_6}$ there is no fragment. We have relabeled the markings of all fragments by $\mathsf{v_0}, \ldots, \mathsf{v_9}$ to make the internals of P anonymous. For each edge of R^*, we define a connection. The connection is calculated from the edges of R^* and the markings.

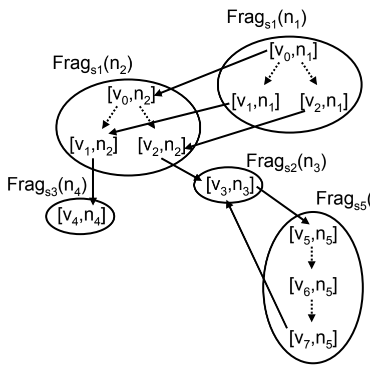
Given a service R, the most permissive strategy R^* for P and the fragments and connections of P, we show in the following how a transition system $P \oplus R$ can be constructed.

2.3 Fragments and Connections for *P*

From the construction of fragments we know that for each state s of R^*, its state space is defined by the fragment Frag_s. Furthermore, for each pair of fragments $\mathit{Frag}_1 \neq \mathit{Frag}_2$, the set of edges with source in Frag_1 and sink in Frag_2 is defined by connection $C_{\mathit{Frag}_1,\mathit{Frag}_2}$. For each fragment Frag, let $\mathit{id}_{\mathit{Frag}}$ denote connection $C_{\mathit{Frag},\mathit{Frag}}$. Then, by Lemma 1, R can only be a strategy for P if R^* (weakly) simulates R.



(a) Requester R



(b) TS_R

Fig. 3. Constructing the state space $P \oplus R$ from the fragments for a given service R. Note that the $\mathsf{?b}$ edge in state n_5 yields $[\mathsf{n}_4, \mathsf{s}_6] \in \rho$. However, as s_6 is not reachable, there is no fragment for state s_6 and hence there is no transition from Frag_5 to Frag_3.

Definition 1 (Construction of TS_R). *Let ϱ be simulation relation between R and R^*. Compose transition system TS_R from the following fragments and connections:*

- $FRAG = \{Frag_s(n) \mid [n, s] \in \varrho\}$,
- $CONN = \{C_{Frag_s, Frag_{s'}}(n, n') \mid [n, s] \in \varrho, [n, x, n'] \in \delta_R \ (x \neq \tau), [s, x, s'] \in \delta_{R^*}$ *(which implies* $[n', s'] \in \varrho)\} \cup \{\, id_{Frag_s}(n, n') \mid [n, s] \in \varrho, [n, \tau, n'] \in \delta_R\}$

The fragment that corresponds to the initial state of R and R^* is unique and it contains the initial state of the resulting transition system TS_R.

In our example, $(\mathsf{n}_1, \mathsf{s}_1)$, $(\mathsf{n}_2, \mathsf{s}_1) \in \varrho$. Thus we add two instances of $\mathsf{Frag}_{\mathsf{s1}}$, i.e. $\mathsf{Frag}_{\mathsf{s1}}(\mathsf{n}_1)$, $\mathsf{Frag}_{\mathsf{s1}}(\mathsf{n}_2)$. As we have transition $[\mathsf{n}_1, \tau, \mathsf{n}_2] \in \delta_R$ in R we add connection $id_{\mathsf{Frag}_{\mathsf{s1}}}(\mathsf{n}_1, \mathsf{n}_2)$. Furthermore, we add fragment $\mathsf{Frag}_{\mathsf{s3}}(\mathsf{n}_4)$ because $(\mathsf{n}_4, \mathsf{s}_3) \in \varrho$. From transition $[\mathsf{n}_2, ?\mathsf{a}, \mathsf{n}_4] \in \delta_R$ in R and $[\mathsf{s}_1, ?\mathsf{a}, \mathsf{s}_3] \in \delta_{R^*}$ we conclude that connection $C_{\mathsf{Frag}_{\mathsf{s1}}, \mathsf{Frag}_{\mathsf{s3}}}(\mathsf{n}_2, \mathsf{n}_4)$ for $?\mathsf{a}$ has to be added. Figure 3(b) shows the resulting state space $P \oplus R$. TS_R contains a livelock (the nodes of $\mathsf{Frag}_{\mathsf{s2}}(\mathsf{n}_3)$ and $\mathsf{Frag}_{\mathsf{s5}}(\mathsf{n}_5)$ have no final node), thus R is no strategy for P.

The resulting transitions system TS_R can be verified for deadlocks and livelocks. Our main result of this paper guarantees that each deadlock and livelock in $P \oplus R$ is preserved in TS_R and vice versa.

Theorem 1. *Let R be a strategy for P and let TS_R be as defined above. Then TS_R is bisimilar to the state space $P \oplus R$.*

3 Deadlock and Livelock-preserving Abstraction

To speed up the model checking run when checking TS_R, we can apply state-of-the-art reduction techniques such as partial-order and symmetry reduction. Besides this, we statically reduce the fragments. This may lead to a smaller TS_R and thus increasing the performance of our approach and, in addition, we need to store smaller fragments in the repository (when publishing P). All abstractions we sketch in the following preserve both deadlocks and livelocks.

For each fragment we compute its strongly connected components (SCCs). It is sufficient to store only SCCs instead of nodes.

To condense the state space of each fragment, we adapt state space condensation rules from [3]. These rules can be applied to each fragment. For example, we can condense the three SCCs in $\mathsf{Frag}_{\mathsf{s5}}$ (see Fig. 1(a)) to a single SCC.

Finally, we can also minimize R, in particular, its τ transitions. For instance, we can apply minimization rules that preserve branching bisimulation. That way, states n_1 and n_2 in Fig. 3(a) could be merged.

4 Conclusion

We have proposed a technique to realize the “find” operation in SOA in case the composed system is required to be free of deadlocks *and* livelocks. We suggest that a service provider publishes a set of state space fragments such that each fragment carries a part of the whole behavior of P. Given a requester R, “find” means to construct the state space $P \oplus R$ from the fragments of P guided by the behavior of R. The resulting state space is checked for deadlocks and livelocks.

Although the space complexity is the product of the state spaces of R and P (in worst case), we assume that applying abstraction techniques results in much smaller states spaces.

We are currently implementing the proposed approach in our analysis tool Fiona [4]. The computed state space $P \oplus R$ can then be checked for deadlocks and livelocks using the model checker LoLA [5]. Future work also includes a case study to validate the strength of the abstraction techniques.

References

1. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings. Volume 4546 of Lecture Notes in Computer Science., Springer-Verlag (2007) 321–341
2. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
3. Juan, E.Y.T., Tsai, J.J.P., Murata, T.: Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules. ACM Trans. on Programming Languages and Systems **20**(5) (1998) 917–979
4. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In Dustdar, S., Fiadeiro, J., Sheth, A., eds.: Fourth International Conference on Business Process Management, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings. Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag (2006) 17–32
5. Schmidt, K.: LoLA: A low level analyser. In Nielsen, M., Simpson, D., eds.: 21st International Conference on Application and Theory of Petri Nets, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding. Number 1825 in Lecture Notes in Computer Science, Aarhus, Denmark, Springer-Verlag (2000) 465–474

WoPeD 2.0 goes BPEL 2.0

Andreas Eckleder[1], Thomas Freytag[2]

[1] Nero AG, Karlsbad, Germany
eckleder@woped.org
[2] University of Cooperative Education (Berufsakademie), Karlsruhe, Germany
freytag@woped.org

Abstract. *WoPeD (Workflow Petrinet Designer) is an easy-to-use, Java-based open source software tool being developed at the University of Cooperative Education, Karlsruhe. WoPeD is able to edit, simulate and analyze workflow nets, providing a useful instrument in particular for research and educational purposes. This paper gives an overview to the current features of the tool, in particular on the newly-added BPEL export and coverability graph visualization capabilities which will be part of the next major release 2.0*

What is WoPeD?

WoPeD is Java-based and supports the class of workflow nets as well as standard place/transition nets. WoPeD is strictly supporting the well-established "van der Aalst" notation [Aal02] and can visualize both structure and dynamics of workflow processes, helping to get a deeper and more intuitive understanding of the underlying theoretical concepts. WoPeD mainly focuses on educational, scientific and publishing purposes in the field of Petri net-based workflow modelling. WoPeD is open source and freely-available. Source code and installer packages are provided via Sourceforge[1], a common platform for the distributed development of free software projects. Several publications have accompanied the emerging development of the software, giving additional information on the underlying architecture [FrL03], on used algorithms [Eck06] and on visualization concepts [FlF06]. A new major release will be released in the fourth quarter of 2008. The rest of this paper gives a brief overview on the most important features of WoPeD, with special focus on new functions which will be part of the new major release 2.0.

Process and resource view editors

The WoPeD editor offers full support for the class of workflow nets including operators, triggers, sub-processes, resource assignments and quantitative parameters like task service times or branching probabilities of XOR-splits. In addition, WoPeD contains a separate graphical resource modelling editor to define resource classes (groups and roles) and their contained resource objects (workflow participants).

[1] http://sourceforge.net/projects/woped

Within the process model, each resource-triggered transition can be associated with one role and one group. The standard file format of WoPeD is PNML [WeK03], allowing model exchange with other Petri net tools. For convenient import into other

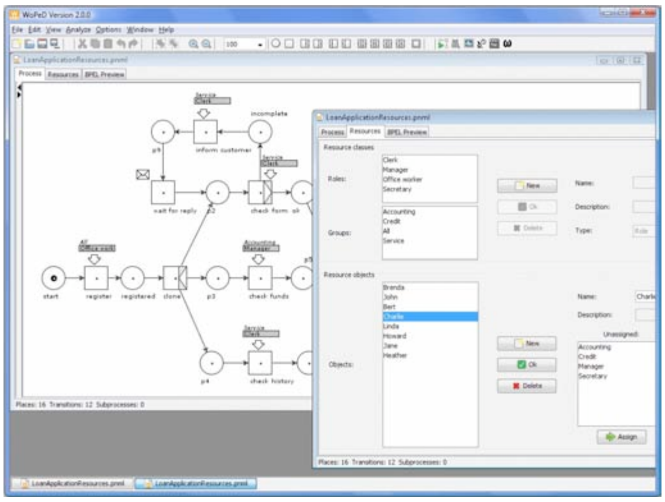


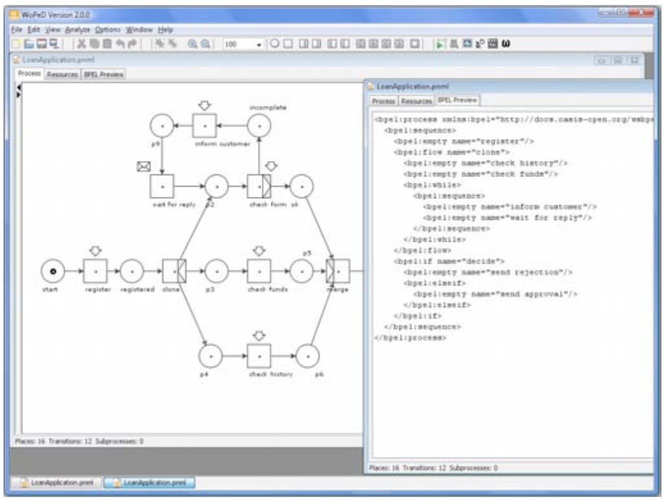
Fig. 1

Process and resource editor

tools supporting PNML, complex operators in "van der Aalst" notation are expanded automatically into their Petri net primitives. WoPeD supports several export interfaces, including JPEG, BMP and PNG graphic formats.

Exporting workflow nets to BPEL

An important new feature of WoPeD 2.0 is exporting well-structured, free-choice workflow nets into the widely-used BPEL format. The process control flow is converted to the associated BPEL constructs and single transitions can be used as placeholders for basic BPEL operations (*assign, invoke, receive, reply, wait*). A



```
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpe
  <bpel:sequence>
    <bpel:empty name="register"/>
    <bpel:flow name="clone">
      <bpel:empty name="check history"/>
      <bpel:empty name="check funds"/>
      <bpel:while>
        <bpel:sequence>
          <bpel:empty name="inform customer"/>
          <bpel:empty name="wait for reply"/>
        </bpel:sequence>
      </bpel:while>
    </bpel:flow>
    <bpel:if name="decide">
      <bpel:empty name="send rejection"/>
      <bpel:elseif>
        <bpel:empty name="send approval"/>
      </bpel:elseif>
    </bpel:if>
  </bpel:sequence>
</bpel:process>
```

Fig. 2

BPEL 2.0 export tab

global namespace is supported for defining state variables which can be used as parameters when interacting with web-services. By this, WoPeD allows the orchestration of arbitrary web services identified by partner links as well as their import from UDDI business registries. The parser used to convert the workflow net control flow into a executable BPEL script is based on the ideas published in [AaL08] and [Las06].

Sound sub-process support

WoPeD allows hierarchical editing of sub-processes. Any transition of a workflow net can be an abstraction of another workflow net, symbolized by a special, double-framed sub-process transition symbol. By this, even large workflow process models can be managed by splitting them up into small portions. As a restriction, WoPeD forces all sub-processes to be workflow nets, such that only subnets with exactly one input and one output place are supported. This has the interesting consequence that

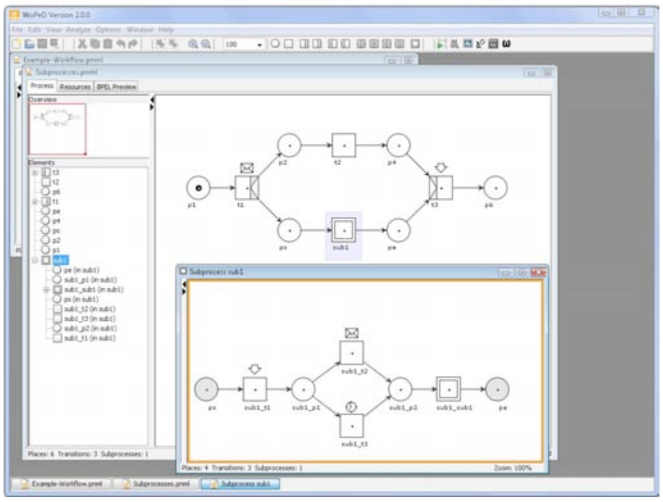


Fig. 3

Sub-process editor and process tree view

most qualitative and quantitative behaviour properties (like e. g. soundness) can be checked locally on sub-process level and the results can be recursively exported to the embedding process levels. Currently, the composition of the analysis results must be done manually. A future version of WoPeD will be able to automate this task by creating a hierarchical analysis report over all sub-processes.

Enhanced simulation control

WoPeD provides an animated token game simulation for navigating through the reachable markings of a workflow net. The new version 2.0 contains an improved interface to navigate both forward and backward, step into or step over sub-processes and automatically proceed to the next conflicting marking. Apart from this, WoPeD allows recording and playback of simulation sessions as well as saving them for later reference. A comfortable "remote-control"-like widget in three different views (standard, compact and "iPod") is provided in order to control all these activities.

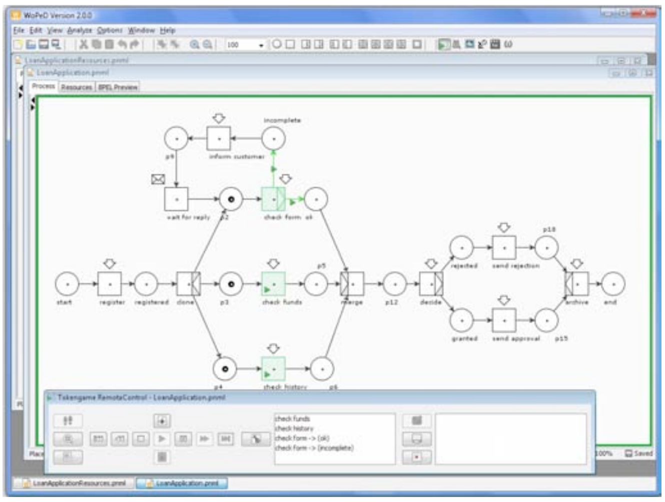
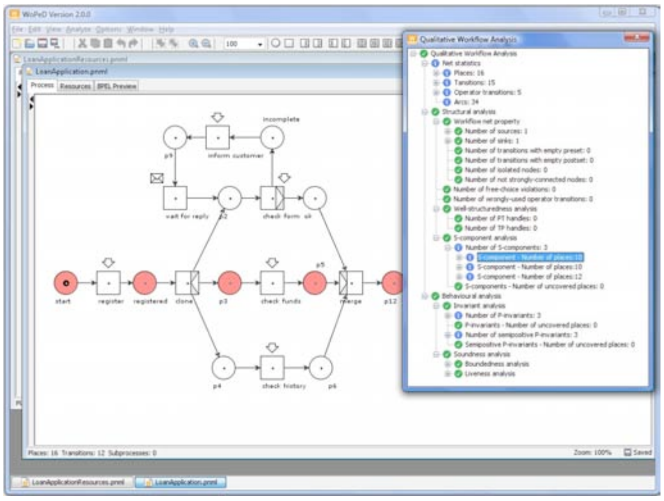


Fig. 4

Simulation "remote control"

Qualitative analysis and visual debugging

WoPeD can analyse a variety of qualitative, soundness-related properties, e. g. free-choice, S-component coverage, well-structuredness, boundedness and liveness. Almost all properties are checked by built-in algorithms, except for some runtime



```
Qualitative Workflow Analysis
  Net statistics
    Places: 16
    Transitions: 15
    Operator transitions: 5
    Arcs: 34
  Structural analysis
    Workflow net property
      Number of sources: 1
      Number of sinks: 1
      Number of transitions with empty preset: 0
      Number of transitions with empty postset: 0
      Number of isolated nodes: 0
      Number of not strongly-connected nodes: 0
    Number of free-choice violations: 0
    Number of wrongly-used operator transitions: 0
    Well-structuredness analysis
      Number of PT handles: 0
      Number of TP handles: 0
    S-component analysis
      Number of S-components: 3
        S-component - Number of places:10
        S-component - Number of places:10
        S-component - Number of places:12
      S-components - Number of uncovered places: 0
  Behavioural analysis
    Invariant analysis
      Number of P-invariants: 3
      P-invariants - Number of uncovered places: 0
      Number of semipositive P-invariants: 3
      Semipositive P-invariants - Number of uncovered places: 0
    Soundness analysis
      Boundedness analysis
      Liveness analysis
```

Fig. 5

Visual debugging of net properties

consuming parts of the soundness check which are computed by using Woflan [VBA01] [Wof08] transparently as an external library. This allows the direct graphical visualization of the analysis results inside the associated workflow net, making WoPeD a powerful visual debugging tool for workflow process definitions.

Automated coverability graph creation

The new release 2.0 of WoPeD implements automatic reachability and coverability graph construction and visualization. Two simple layout algorithms are implemented

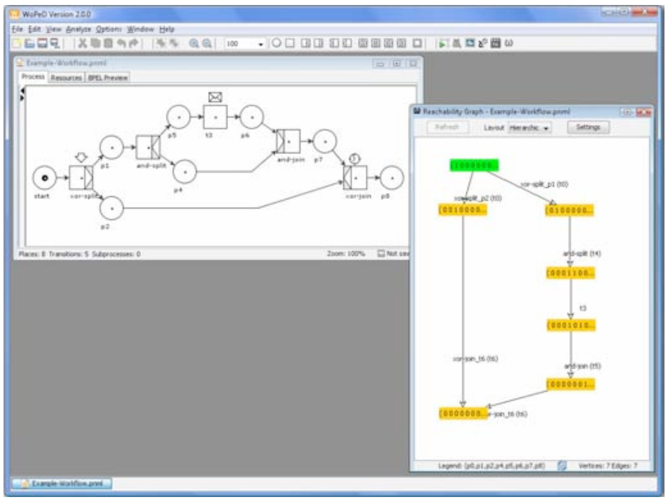
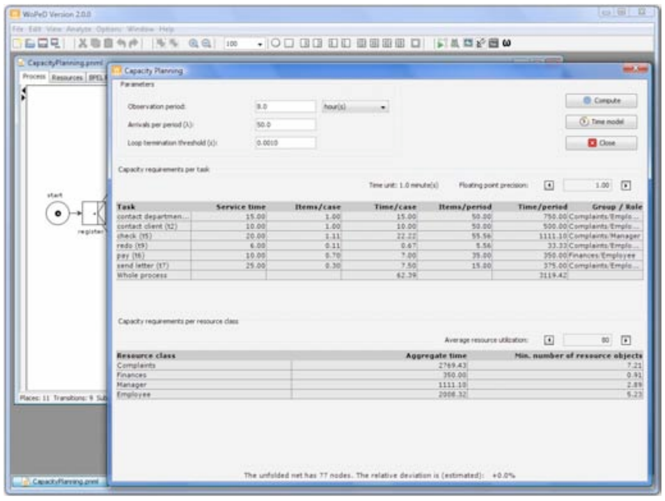


Fig. 6

Coverability graph visualization

to allow a visual representation. To facilitate the creation of visually appealing graph representations, the software allows users to adjust node positions according to their needs. Once displayed in a satisfying way, the coverability graph can also be exported to the most common graphics formats such as JPEG, BMP and PNG. By this, WoPeD can be used to create sample graphs e. g. for lecture material or other publications.



Parameters

Observation period: 8.0 hour(s)

Arrivals per period (λ): 50.0

Loop termination threshold (ε): 0.0010

Capacity requirements per task

Time unit: 1.0 minute(s) Floating point precision: 1.00

Task	Service time	Items/case	Time/case	Items/period	Time/period	Group / Role
contact departmen...	15.00	1.00	15.00	50.00	750.00	Complaints/Emplo...
contact client (t2)	10.00	1.00	10.00	50.00	500.00	Complaints/Emplo...
check (t5)	20.00	1.11	22.22	55.56	1111.10	Complaints/Manager
redo (t9)	6.00	0.11	0.67	5.56	33.33	Complaints/Emplo...
pay (t6)	10.00	0.70	7.00	35.00	350.00	Finances/Employee
send letter (t7)	25.00	0.30	7.50	15.00	375.00	Complaints/Emplo...
Whole process			62.39		3119.42	

Capacity requirements per resource class

Average resource utilization: 80

Resource class	Aggregate time	Min. number of resource objects
Complaints	2769.43	7.21
Finances	350.00	0.91
Manager	1111.10	2.89
Employee	2008.32	5.23

The unfolded net has 77 nodes. The relative deviation is (estimated): +0.0%

Fig. 7

Capacity planning report

Quantitative analysis and capacity planning

WoPeD is able to store and visualize an average service time value with each resource-trigged transition and an average branching probability with each outgoing arc of an implicit or explicit XOR-split operator. Based on the role/group assignment to all contained tasks, this allows the computation of a capacity planning table derived from both process and resource model, assigning each transition the expected number of work items per case, and each resource class the minimum required number of members under a given resource utilization rate. The algorithm to compute the number of work items per case is based on a special sort of net unfolding which is capable to approximate possibly infinite loop behaviour.

Conclusion and outlook

WoPeD is an evolving software tool. The editing component supports a process model view as well as a resource model view and a functional view (currently BPEL code). WoPeD strictly supports the original workflow net notation and contains algorithms for checking qualitative properties (soundness) as well as quantitative properties (capacity planning). By this, WoPeD is an instrument for "blended learning" in the context of teaching and publishing in the area of workflow management and process analysis. Future development will focus on an enhanced resource model editor, a more powerful coverability graph visualization and more additional process debugging and analysis functions. For further information, including announcements of new features, download links, screenshots and documentation, please refer to the website [WoP08].

References

[AaH02] W. M. P. van der Aalst, K. van Hee. Workflow Management – Models, Methods and Systems. MIT Press, Cambridge, 2002.

[AaL08] W. M. P. van der Aalst, K. B. Lassen: Translating unstructured workflow processes to readable BPEL: Theory and implementation. Information & Software Technology 50(3): 131-159 (2008).

[FlF06] C. Flender, T. Freytag - Visualizing the Soundness of Workflow Nets AWPN Workshop, Hamburg, 2006.

[FrL03] T. Freytag, S. I. Landes. PWFtool – a Petri net workflow modelling environment. Proceedings of the Workshop "Algorithmen und Werkzeuge für Petrinetze (AWPN), Eichstätt, 2003.

[Las06] K. B. Lassen, W. M. P. van der Aalst: WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. OTM Conferences (1): 127-144 (2006).

[VBA01] H. M. W. Verbeek, T. Basten, W. M. P. van der Aalst: Diagnosing Workflow Processes using Woflan. Computer Journal 44(4): 246-279 (2001).

[WeK03] M. Weber, E. Kindler. The Petri Net Markup Language. In: H. Ehrig, W. Reisig, G. Rozenberg, H. Weger (Eds.). Petri Net Technology for Communication Based Systems, LNCS 2472, Springer 2003.

[Wof08] The Woflan Homepage. is.tm.tue.nl/research/woflan.htm, 2008.

Synthesis of Petri Nets from Infinite Partial Languages with VipTool

Robin Bergenthum and Sebastian Mauser

Catholic University of Eichstätt-Ingolstadt, Germany
prename.name@ku-eichstaett.de

Abstract. In this paper we show an implementation of an algorithm to synthesize a place/transition Petri net (p/t-net) from a possibly infinite partial language, which is given by a term over a finite set of labelled partial orders (LPOs).
The implementation is integrated as an extension of our Petri net toolset VipTool. The new extension comprises two plug-ins. The first plug-in offers editing features to specify term based partial languages. The specification of terms is graphically supported by a visualization similar to structograms as well as by a representation in the form of UML activity diagrams. The second new plug-in provides the algorithmic computation of a p/t-net synthesized from a term specification. This algorithm is in principle based on ideas already presented in the paper [4], but a so called separation computation is applied instead of the basis representation used in [4].

1 Introduction

Synthesis in the field of Petri net theory means algorithmic construction of a Petri net satisfying a given behavioural description. Often labelled partial orders (LPOs) are considered the most natural model to describe the non-sequential behaviour of Petri nets. An LPO represents a run of a place/transition Petri net (p/t-net) if it is enabled w.r.t the net in the sense that the events of the LPO modelling transition occurrences can fire in the net respecting the concurrency and dependency relations given by the LPO. In [4] we presented an algorithm to synthesize a finite unlabelled p/t-net from a partial language, i.e. a set of LPOs, which is given in a term based representation. It was shown that the set of runs of the synthesized net coincides with the set of LPOs represented by the term – if such a net exists. A term of LPOs is built from a finite alphabet of LPOs and composition operators for union, parallel composition, sequential composition and iteration. Due to the iteration operator the partial language represented by such term may be infinite. The synthesis approach in [4] is based on the theory of regions. Each transition of the synthesized net is given by a label appearing in the term, and the places of the net are computed by so called token flow regions. Since the set of all regions is infinite, the synthesis algorithm calculates a finite set of so called basis regions.

It was shown in [3] that such basis representation is often inappropriate for practical applications, because the set of places corresponding to basis regions is usually very large causing unreadable nets. Therefore, in this paper we change the synthesis algorithm from [4] by computing so called separating regions, which proved to yield good synthesis results [3]. Moreover we use the region type called transition regions instead of the token flow regions considered in [4]. But this is only a design decision, because we showed in [3] that both region types have advantages in different settings. While exchanging the region type of token flow regions by transition regions only requires minor changes of the synthesis algorithm, it is difficult to use the principle of computing a separating representation of all regions instead of a basis representation. The problem in particular lies in the infinity of the considered partial languages. Tackling this problem to get a synthesis algorithm from terms of LPOs which uses separating regions is the main theoretical contribution of this paper.

On the practical side we implemented this algorithm as an extension of our toolset VipTool [1] (viptool.ku-eichstaett.de). VipTool offers a user-friendly framework for integrating plug-ins. The focus of VipTool is on partial order behaviour of Petri nets. In particular, VipTool already offers functionalities for editing, visualizing and storing p/t-nets and LPOs. The new extension comprises a plug-in to specify terms of LPOs. It allows to compose stored LPOs by respective composition operators. The composition of LPOs is graphically supported by a visualization of the term in a style analogous to structograms. This representation of such terms was already suggested in [2] as a nice possibility to compose LPOs. The plug-in also offers an alternative visualization of terms of LPOs in the form of UML activity diagrams which might be more intuitive for some users. The second plug-in is an implementation of the actual synthesis algorithm developed in this paper. It computes a p/t-net from a term of LPOs which can then be layouted and displayed by VipTool.

2 Synthesis Algorithm

In this section the new synthesis algorithm is described. Due to lack of space, we omit formal details here, and try to give a high-level explanation of the algorithm. We provide a toy example concerning a workflow of processing a claim in an insurance company.

The input to the synthesis algorithm is a term over a finite alphabet of LPOs serving as building components. Terms are constructed inductively by iteration (*), parallel composition (||), sequential composition (;) and union (+). A term defines a possibly infinite set of LPOs, called the partial language of the term, by combining the LPO components according to the composition operators. For example the infinite partial language of the term *Registration; (((PosEvaluation; ((Queries)*); Payment) + NegEvaluation) || Reserves)* build of the LPOs given in Figure 1 is illustrated in Figure 2. For formal definitions see [4].

The aim now is to compute a p/t-net, such that the set of enabled LPOs of the net coincides with the partial language of the specified term (or more precisely with its prefix- and sequentialization closure) - if such net exists. The synthesis algorithm proposed in this paper follows standard techniques known from the theory of regions. It roughly works as follows: The set of transitions of the synthesized net is the finite set of labels appearing in the term. One restricts the behaviour of this net by creating causal dependencies between the transitions through addition of places. Places are defined by their initial marking and the arc weights connecting them to transitions. Two kinds of places can be distinguished. In the case that there is an LPO in the partial language of the term which is not enabled in the net which has only the one considered place, this place restricts the behaviour too much. Such places are non-feasible. In the other

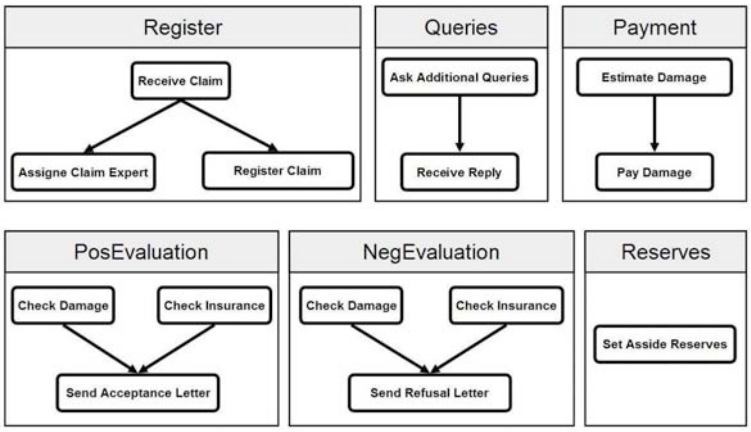


Fig. 1. Example LPOs modelling parts of a claim handling process in an insurance company.

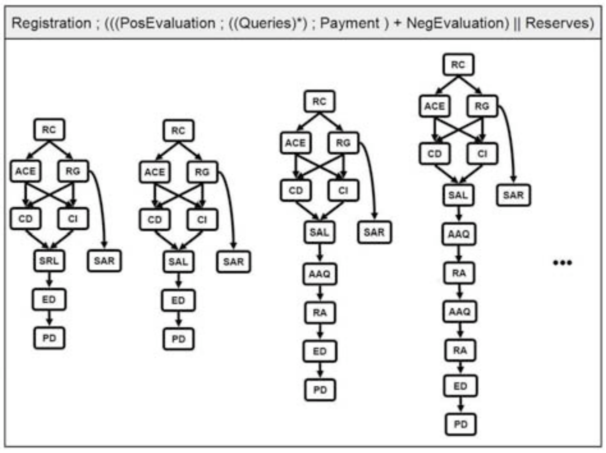

Registration ; (((PosEvaluation ; ((Queries)*) ; Payment) + NegEvaluation) || Reserves)
RC
ACE
RG
CD
CI
SRL
SAL
SAR
AAQ
RA
ED
PD
...

Fig. 2. Example term over the LPOs given in Figure 1 and the corresponding infinte partial language (transition names are abbreviated).

case, the considered place is feasible. The set of feasible places corresponds to the set of regions of the term. The definition of regions considered in this paper is similar to the one developed in [4]. But the idea of considering token flows is replaced by using so called transition regions (compare [3]). Since in general there are infinitely many regions, in [4] a finite set of so called basis regions representing all feasible places is computed. The approach in this paper is different. The aim is to iteratively compute feasible places prohibiting LPOs not in the language of the term from being enabled, such that after a finite number of steps all behaviour not given by the term is not any more possible in the calculated net. The basic idea is to append events to LPOs given by the language of the term such that the resulting behaviour, called a wrong continuation of the term, is not specified by the term. Then for each wrong continuation a region is computed such that the corresponding feasible place separates the wrong continuation (compare [3]), if such a place exists. In the positive case the place is added to the net and the next wrong continuation is considered, in the negative case it is directly proceeded with the next wrong continuation. This solves the synthesis problem, because the finite set of wrong continuations is defined in such a way that, if all wrong continuations are separated, then the infinite set of all LPOs not in the language of the term is prohibited, or it is not possible to prohibit all such behaviour. The problem is first to define the notion of regions of a term appropriately and second to define wrong continuations of a term in such a way that the set of wrong continuations is finite, although the language of a term may be infinite.

The idea in [4] to define regions of a term is to consider a finite set of representation-LPOs R and a finite set of iteration-LPOs I (see Figure 3 for an example). The region definition ensures that the LPOs in R are enabled, and that the LPOs in I can be iterated, which means that if such LPO is enabled in a marking then it is again enabled after having fired all events of the LPO. Consequently the LPOs in R are enabled w.r.t. a place corresponding to a region (i.e. the place is feasible w.r.t. R) and the LPOs in I produce at least as many tokens in a place corresponding to a region as they consume. It is shown in [4] that a place is feasible w.r.t. the language of a term if and only if it satisfies these constraints. In this paper we consider regions of a term by applying these constraints to transition regions. Then similar as in [4] the set of feasible places of the language of a term corresponds to the set of regions of the term where the set of regions is given by the integer solutions of an inequality system.

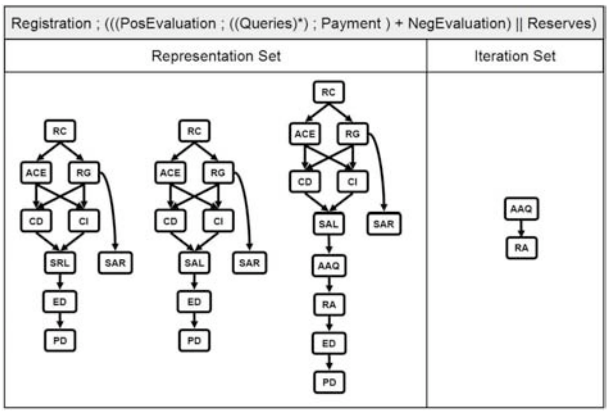

Registration ; (((PosEvaluation ; ((Queries)*) ; Payment) + NegEvaluation) || Reserves)
Representation Set
Iteration Set
RC
ACE
RG
CD
CI
SRL
SAL
SAR
AAQ
RA
ED
PD

Fig. 3. Example term of Figure 2 and the corresponding finite representation and iteration sets.

Having defined regions we have to tackle the problem of defining wrong continuations. The basic idea is to use the notion of wrong continuations of a finite partial language developed in [3] on the finite set R. Note that due to iteration it may happen that a wrong continuation of an R-LPO is an element of the partial language of the term, but this does not matter, because in this case there is of course no feasible place prohibiting this wrong continuation and thus the step is skipped. The aim is that if we separate all wrong continuations of R, then also all wrong continuations of LPOs that can be generated by iterating the I-part of an R-LPO are separated. More precisely, a feasible place separating a wrong continuation of an R-LPO should also separate the respective wrong continuation of an LPO arising from this R-LPO through iteration of I-LPOs. But we found examples showing that this is not automatically guaranteed, because separating places may be filled with tokens by iterating I-LPOs, such that these places do not any more separate behaviour after enough iterations. Therefore, we need some constraints ensuring that iterating certain I-LPOs does not pump up the tokens in a separating place, i.e. such I-LPOs are not allowed to produce more tokens in the place than they consume from the place. But there are examples showing that we cannot introduce such restriction for each I-LPO and each separating place, because sometimes only an unsafe place can prohibit a certain wrong continuation. The solution is to identify for each feasible place separating a wrong continuation of an R-LPO an appropriate subset of I-LPOs, for which we consider a ”non-pump-up” constraint. This subset is roughly speaking given by the I-LPOs that can occur before the wrong continuation within the R-LPO, because each such I-LPO could otherwise pump up the place so that the wrong continuation is not separated after a certain number of iterations of the I-LPO.

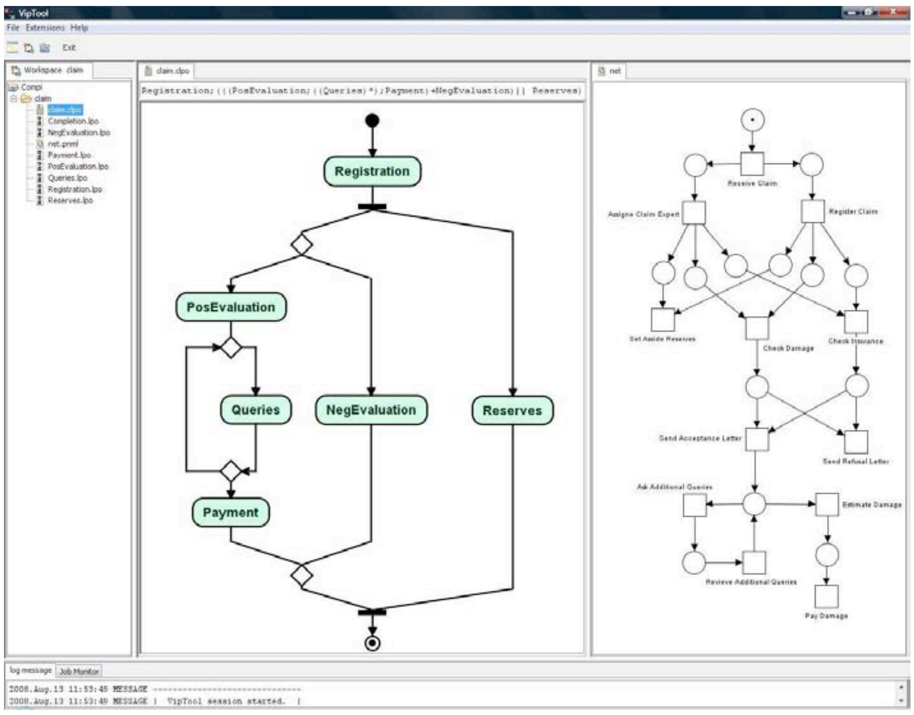
The algorithm described semi-formally in this section synthesizes a p/t-net from a term of LPOs. For the term of Figure 2 the resulting net is shown on the right part of Figure 4. Without providing a formal proof in this workshop paper, we claim that if there is a p/t-net having the partial language of the given term as its set of enabled LPOs, then the computed net is such net. The decidability if the computed net actually has the given behaviour is an open problem (see also [4]).

3 Implementation

We implemented the algorithm presented in the last section as an extension of our toolset VipTool [1]. VipTool was originally designed as a tool for modelling, simulation, validation and verification of business processes using (partial order behaviour of) Petri nets. It now offers a flexible xml-based open plug-in architecture to integrate methods concerned with causality and concurrency modelled by partially ordered runs of Petri nets. In particular, it already provides plug-ins to synthesize p/t-nets from finite partial languages. The new extension to synthesize p/t-nets from terms of LPOs now offers two important advancements: On the one hand it is for the first time possible to compute nets from infinite sets of LPOs (finitely represented by terms), and on the other hand terms allow a modular specification of the input partial language. The support of modularity and the possibility to specify infinite behaviour are two innovative steps towards practical applicability of our synthesis framework in industrial settings, where our focus remains on modelling of business processes.

The new synthesis extension can nicely build on VipTool’s existing plug-ins for editing, visualizing and storing p/t-nets and LPOs. A first new plug-in of VipTool supports the design of terms of LPOs. Using LPOs drawn with VipTool and stored as xml-files, the plug-in provides an editor to specify terms over such LPOs. The composition of single LPOs to terms of LPOs by using the provided composition operators is supported by two graphical views. The first visualization illustrates the inductive composition of the LPOs in a natural way by building a block structure illustrating the composition operators between the blocks (as shown in [2]). This is very similar to the visualization of algorithms by structograms. The second visualization translates the term structure into an UML activity diagram (see Figure 4). The alphabet of single LPOs underlying the term form the activities of the diagram. In this sense, they have to be interpreted as abstract activities that are refined by a behavioural description through the respective LPOs. The sequential composition operator determines the path dependencies between the LPO-activities. The parallel and alternative composition operators yield balanced parallel and alternative splits and joins. The iteration operator is implemented as a respective cyclic structure. Having designed a specification of partial order behaviour of Petri nets by a term, a description of the term of LPOs is stored in an xml-file.

A second plug-in loads such xml-file and xml-files of the LPOs occurring in the term. It computes a p/t-net from the term specification by applying the described algorithm. The resulting net is stored as a pnml-file, which can be loaded, visualized, layouted and analyzed by already existing plug-ins of VipTool. Figure 4 shows that in our running example the computed net nicely models the workflow specified by the considered term.



```
Registration;(((PosEvaluation;((Queries)*);Payment)+NegEvaluation)|| Reserves)
```

```
2008.Aug.13 11:53:49 MESSAGE ----------------------------
2008.Aug.13 11:53:49 MESSAGE |  VipTool session started.  |
```

Fig. 4. Screenshoot of VipTool showing the example term of Figure 2 represented as an activity diagram and the net resulting from the synthesis algorithm.

References

1. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Scenarios with Viptool. In *ICATPN 2008, LNCS 5062*, pages 388–398.
2. R. Bergenthum, J. Desel, and S. Mauser. Synthesis of Petri Nets for Business Process Design. In *Proceedings of workshop Verhaltensmodellierung: Best Practices und neue Erkenntnisse @Modellierung 2008*.
3. R. Bergenthum and S. Mauser. Comparison of Different Algorithms to Synthesize a Petri Net from a Partial Language. In *Proceedings of workshop CHINA @ICATPN 2008*.
4. R. Lorenz, R. Bergenthum, S. Mauser, and J. Desel. Synthesis of Petri Nets from Infinite Partial Languages. In *Proceedings of ACSD 2008, IEEE.*

Adding Runtime Net Manipulation Features to MULANVIEWER

Jan Schlüter, Lawrence Cabac, Daniel Moldt

University of Hamburg, Department of Computer Science,
Vogt-Kölln-Str. 30, 22527 Hamburg
`http://www.informatik.uni-hamburg.de/TGI/`

Abstract. MULANVIEWER, a MULAN inspection tool, is focused on gathering information from a Petri net-based multi-agent system and greatly helps finding bugs, but fixing them is overly time-intensive. We overcome this limitation by extending MULANVIEWER with runtime net manipulation features. To do so, we analyze a typical debugging cycle, point out bottlenecks and implement the most promising additions. The new features considerably accelerate the identification and fixing of bugs frequently encountered in MULAN applications. Overall the enhancements complement MULANVIEWER's features to navigate large and complex Petri net implementations by adding manipulation capabilities.

1 Introduction

The PAOSE approach is based on high-level Petri nets forming an agent-oriented structure, embedding other successful techniques, such as Java and UML. The development process is highly complex. However, it is supported by a set of powerful tools and we improve the development process by continuous investigation of techniques, tools and methods. In our last project, the time consuming debugging process was one of our targets. In Section 2 we sketch the PAOSE development setting, highlight the challenges in debugging agent systems and describe current solutions within MULAN. Subsequently, we identify debugging bottlenecks by timing common tasks during PAOSE debugging processes and suitably extend MULANVIEWER in Section 3. Section 4 closes with a short summary and an outlook of future work.

2 Developing Multi-Agent Systems with Mulan

Before tackling our goal, we will look at the existing work we can build on: the current development environment, the challenges during debugging multi-agent systems and the toolset meeting them.

2.1 The PAOSE Development Environment

The multi-agent system development environment used for Petri net-based agent-oriented software engineering is based on RENEW (Reference net workshop [4]).

Renew provides a graphical user interface for creating and editing Petri nets and a simulator supporting different formalisms. In the Java net formalism, every token can be a reference to another net or an arbitrary Java object. Nets can communicate with embedded nets through synchronous channels and directly work with Java objects using our Java-based inscription language. During the simulation, Renew visualizes the token game, allowing its user to interactively pause and continue the simulation, fire transitions manually, set breakpoints on transitions or places and inspect the token in detail.

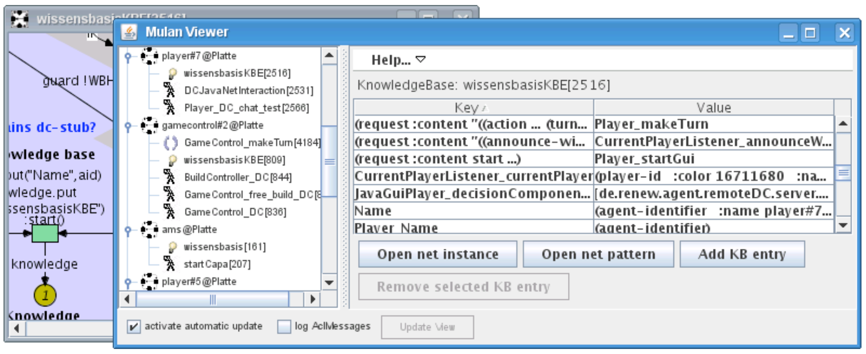
Mulan/Capa is our FIPA-compliant framework for developing multi-agent systems (MAS) based on reference nets and Java. For Mulan, we identify three orthogonal views on a MAS [1]: structure, behavior and terminology. The framework respects this by separating the agents and their knowledge from their behavior, which is encapsulated in protocol nets, and from the Java-based ontology. The implementation of Mulan/Capa extensively uses the nets-within-nets paradigm: The system infrastructure net holds all platform nets, providing global communication and agent mobilization services. Each platform net hosts a number of agents, which can use the platform to exchange messages. An agent, in turn, can be a reference net, too. The default agent holds a declarative knowledge base and *active knowledge* encapsulated in decision component nets as well as a protocol factory, which instantiates protocol nets in reaction to incoming messages based on associations in the knowledge base (reactive behavior) or through internal triggers (proactive behavior). Protocol nets are reference nets defining a structured schedule of internal actions (knowledge base and decision component access) and external actions (messages send out and received). A protocol usually corresponds to a column in an interaction diagram defining the interactions taking place with one or multiple other agents (roles).

2.2 Challenges in Debugging Multi-Agent Systems

Apart from the common pitfalls [6] shared with object-oriented software development, developers of multi-agent systems have to cope with some specific problems, the most important one being that "[m]ultiagent systems tend to lack any central control" [7, p.3]: The system consists of autonomous, possibly distributed agents acting and interacting concurrently without any global instance controlling it. This imposes a number of challenges to be overcome during debugging a MAS. Due to the dependencies between agents, locally inspecting them often does not suffice to track the source of a problem, but the complexity of the system and the lack of a central instance makes it difficult to gain a global overview of the system's state [2]. Furthermore, the concurrent processes within the system may induce nondeterministic behavior [5], preventing the clear reproduction of observed problems.

2.3 Solutions in Mulan

Renew principally provides means to inspect a running multi-agent system, gaining insight into platforms, its agents and their states in terms of net in-



KnowledgeBase: wissensbasisKBE[2516]

Key	Value
(request :content "((action ... (turn...	Player_makeTurn
(request :content "((announce-wi...	CurrentPlayerListener_announceW...
(request :content start ...)	Player_startGui
CurrentPlayerListener_currentPlayer	(player-id :color 16711680 :na...
JavaGuiPlayer_decisionComponen...	[de.renew.agent.remoteDC.server....
Name	(agent-identifier :name player#7...
Player_Name	(agent-identifier)

[✓] activate automatic update
[ ] log AclMessages

Fig. 1. MulanViewer displaying an agent's knowledge base

stances. However, actually finding a certain agent's knowledge base or a particular protocol instance within the running system requires navigating a deep hierarchy of heterogeneous nested nets. This may often be confusing and time-consuming and makes it difficult to grasp a thorough overview about what is happening. The inspection tool called MulanViewer [3,2] fills this gap. Generally spoken, this tool enables its user to easily access the tokens of interesting places and (some) transitions in the complex net system. It does so by browsing the net structure, registering for change notifications at the Renew simulation engine and building a hierarchical model of the system's accumulated state, i.e. the platforms, agents, their knowledge bases, decision components and protocols. This model is visualized in a graphical user interface providing an overview of the active components in the MAS as well as adequate views into these components such as a table listing an agent's declarative knowledge (see Figure 1). For investigation on a lower level MulanViewer allows opening a component's corresponding net in Renew.

The two tools MulanViewer and Renew form a powerful inspection toolset for multi-agent systems implemented in Mulan/Capa. As the system's components are reference nets, most types of errors lead to a transition not being able to fire, which in turn leads to a protocol being blocked. Blocked protocols can easily be found in MulanViewer's overview. Renew's visualization of the protocol net instance allows to quickly locate the problematic transition: The developer just has to look for the lifeline token in the protocol's schedule of actions. The following transition's inscription reveals whether the protocol is blocked because of a missing knowledge base entry, an erroneous decision component channel or other unmet preconditions.

3 Extending MulanViewer

While finding the problem in a MAS is rather easy, fixing it is tedious. As Renew does not provide a way to change a net or its tokens during simulation – which, by the way, might introduce more problems than it solves – developers have to stop the MAS, modify the nets or the agent's initial knowledge base and recompile the project, leading to an overly time-consuming debugging cycle.

We propose that extending the toolset with manipulation features for a running MAS would greatly optimize the process of debugging: Modifying a running MAS dynamically until it works should be faster than modifying the static MAS code, recompiling and restarting it each time. To find out which kinds of features particularly help saving time, we analyze a typical debugging cycle to identify bottlenecks. Subsequently, we extend our toolset to eliminate these bottlenecks by providing means to carefully modify net instances during the simulation.

3.1 Identifying Debugging Bottlenecks

Reflecting on a project course held in Winter 2007/2008, we analyze how the tools described in Section 2.3 are usually used to resolve problems in a Mulan/Capa MAS in order to find the most time-intensive steps. We use a MAS implementing the German board game "Siedler", running on a 2.8 GHz dual core machine with 3 GB working memory to take the time for the tasks. Figure 2 depicts our results: a Petri net modeling the dependencies between debugging tasks and the outcomes of tests.

To be able to debug a component of the MAS, its developers have to ensure they have the latest versions of the project, compile it and bring the MAS into a state that uses their component to see whether it works as intended (steps 1 to 5). Although we assumed the project can be quickly retrieved from a shared repository and the developers can easily activate their component (here: a protocol of an interaction), these steps take up to three minutes.

As we can see in the debugging cycle, these steps have to be repeated on each knowledge base, protocol or decision component related problem. Knowledge base problems comprise an additional bottleneck: The design artifact cannot be merged, thus distributed manipulation synchronized through a repository is not possible, yet.

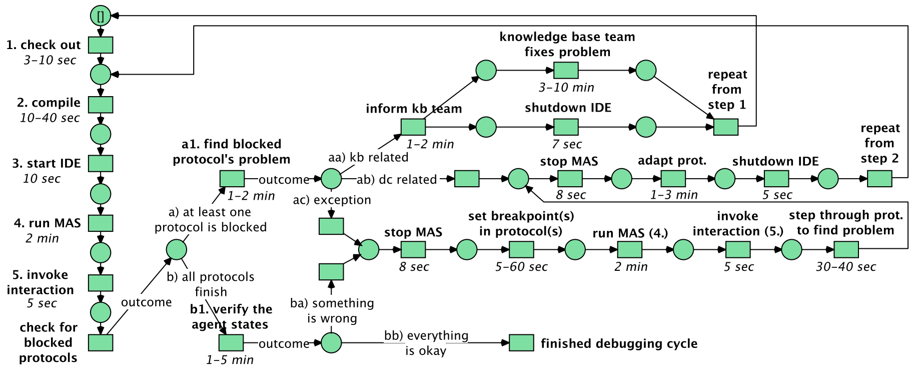


Fig. 2. Petri net modeling a typical Paose debug cycle.

3.2 Implementing Manipulation Features

Although the debugging cycle could be sped up by minimizing the time needed for the repeated steps (most notably steps 1 to 5), this would not change its problematic *structure*. Improving the knowledge base editor to eliminate the dependency on the knowledge base team would arguably help, but we would rather like to completely remove the need for changing the initial knowledge base content just to continue testing. In the same line, decision component and protocol problems should be solvable in the running system.

A general solution for this would be extending Renew to allow editing a net's structure during a simulation, automatically updating all of the net's instances (*Hot Code Replacement*), and to allow modification of a token (*Token Injection*) – this way all problems in knowledge bases, protocols and decision components could be fixed or at least worked around on the fly without needing to restart the system. However, this would require substantial changes to Renew, which are outside the scope of this work. Instead, we will cope with the different types of problems separately, trying to find and implement solutions to fix the most common problems by carefully modifying the running system.

The most usual knowledge base related problems are missing or misspelled entries, causing protocols or decision components accessing them to be blocked. To resolve them, we extend MulanViewer's knowledge base view with capabilities to interactively add, change and remove knowledge base entries. MulanViewer does not directly modify the knowledge base net, but asks Renew's simulation engine to fire the knowledge accessing transitions on the agent's behalf, thus avoiding concurrency problems.

For the specific, but frequent case of erroneous message-to-protocol associations merely fixing the entry often does not suffice: When the problem is spotted by a developer, it usually is because the agent could not interpret a message and suspended an interaction. After correcting the entry, recreating the state of the MAS prior to the misinterpretation may require a restart, which we want to avoid. So we extend MulanViewer to allow moving a message from the protocol factory's *not understood* place to the place of incoming messages, forcing the agent to reinterpret the message without the other agent needing to resend it.

In the case of missing or misnamed DC channels, being able to modify the affected agent's DC certainly would provide the greatest relief. However, as this is out of scope, we implement a feature that loads additional DCs into an agent by instantiating them and injecting them into the agent net. This way misspelled and missing channels can be provided as wrapper and stub channels in a new DC, which can be drawn and loaded by the developer on the fly.

As an all-round solution for circumventing problems during debugging a particular component that are caused by another agent not acting as supposed, we extend MulanViewer to enable developers to start a protocol interactively in the context of an agent. While this does not solve the real problem, it allows a developer to finish testing the respective component independently of the other agent's bugs.

4 Conclusion

We evolved MulanViewer from a mere monitoring tool into an inspection tool capable of manipulating a Mulan multi-agent system during runtime by directly working on the underlying net instances. By this, we reduced the time needed for fixing particularly common problems found during the debugging phase from about 3 minutes to less than a minute, which was our main source of motivation. Apart from that, the new manipulation features help testing a single MAS component outside its usual environment, partly eliminating the dependency between developers when working in a team.

Future work will include additional manipulation features and tool support for repeated tasks in order to further accelerate the debugging cycle. Additionally, we will extend MulanViewer to allow its users to apply runtime manipulations – like editing the knowledge base – to the static code, making changes persistent. As our tool now can considerably interfere with a running MAS, we need to develop a security model, allowing a system or its components to be protected from being manipulated. The MulanViewer idea of directly accessing particular places and transitions provokes a strong dependence on the Mulan implementation anyway, so we plan to rework the architecture, shifting as much responsibility as possible to the MAS, allowing it to determine which and how much information may be published and which states may be modified from outside.

References

1. Lawrence Cabac, Till Dörges, Michael Duvigneau, Christine Reese, and Matthias Wester-Ebbinghaus. Application development with Mulan. In Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom, and Rémi Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 145–159, Siedlce, Poland, June 2007. Akademia Podlaska.
2. Lawrence Cabac, Till Dörges, and Heiko Rölke. A monitoring toolset for Petri net-based agent-oriented software engineering. In Rüdiger Valk and Kees M. Valk van Hee, editors, *29th International Conference on Application and Theory of Petri Nets, Xi'an, China*, volume 5062, pages 399–408, June 2008.
3. Timo Carl. Evaluation und beispielhafte Erweiterung einer referenznetzbasierten Agentenumgebung. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, August 2003.
4. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – the Reference Net Workshop. Available at: http://www.renew.de/, May 2006. Release 2.1.
5. David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *AAMAS*, pages 960–967. ACM, 2002.
6. Bruce F. Webster. *Pitfalls of object-oriented development.* M & T Books, New York, NY, USA, 1995.
7. M.J. Wooldridge and N.R. Jennings. Software engineering with agents: pitfalls and pratfalls. *Internet Computing, IEEE*, 3(3):20–27, May/Jun 1999.

ImageNetDiff: A Visual Aid to Support the Discovery of Differences in Petri Nets

Lawrence Cabac, Jan Schlüter

University of Hamburg, Department of Computer Science,
Vogt-Kölln-Str. 30, 22527 Hamburg
`http://www.informatik.uni-hamburg.de/TGI`

Abstract In this paper we propose a method and present a tool as plugin for Renew that supports the process of discovery of differences in possibly conflicting versions of Petri net code. The method uses the image representaion of the net graph and compares the pixels of the exported Petri nets. The tool uses the image processing of ImageMagick. An open source graphical tool kit, which is available on all common operating systems.

Keywords: high-level Petri nets, nets-within-nets, reference nets, net components, Renew, modeling, agents, multi-agent systems,

1 Introduction

During development of large applications or models with Petri nets developers frequently encounter different and/or conflicting versions of the code base artifacts. Especially in shared projects where Petri net code is shared through source code management systems (SCM) such as the Concurrent Versions System (CVS) or Subversion conflicts frequently appear and have to be resolved manually by the developer. In the evaluation of the code (Petri nets) the main problem is the identification of the syntactical differences or equalities. However, on the one hand formally it is very hard to verify graph equality and even harder to determine the minimum of parts that are different. The graphical representation, on the other hand, may contain valuable hints for the mentioned problems but may also differ without change in the syntax. The merging of changes is usually a manual task, even if only different parts of the nets have been modified. In contrast, when text-based source code is used, merging of non-conflicting concurrent changes is possible. To our knowledge no tools exist so far that manages the merging to some extend or even support the developer in this task. Even if a string representation of the net code exists, usually this code is not handleable by common tools such as *diff* [2] (or windiff).

In this paper we propose a simple but efficient method that can simplify the task of the discovery of differences under certain conditions. To this means we exploit the graphical representation of the nets and transfer the problem to finding differences in the visual image of the Petri nets. We also present an implementation of the method as plugin for Renew [4,5]. In Section 2 we describe the method, its implementation and integration within Renew. Section 3 presents an example to illustrate the method and tool.

2 Discovery of Net Differences

The development of models within development groups frequently leads to conflicting models. Even if the system model is decomposable in many parts, still the problem persists – as with all source code – that within one design artifact (Petri net) several changes can occur concurrently and have to be merged. In this situation two tasks have to be performed. First, the differences have to be identified. Second, the changes have to be included. For Petri nets these tasks usually have to be performed manually.[1] We propose that tool support for the discovery of net differences can accelerate the development of net system models significantly.

2.1 Scenarios

We can distinguish at least two different scenarios in which the tool can be utilized: the *similarity check* and the *difference discovery*. In the similarity check a developer does not know, whether two Petri nets or two versions of the Petri net own the same code (are syntactically/semantically equal but may differ in visually). For text-based code exist code beautifiers that manage to unify the style of code as a preparation for the differences tools. Often net elements or text inscriptions have been moved in the image by another developer and this has been committed to the repository resulting in a conflict. If the nets (or the net versions) contain only small differences (e.g. only one node has been moved) the ImageNetDiff image will show instantly that the nets are syntactically equal. The checking of the equality of the nets is thus reduced to the checking of the graphically differing parts.

In the difference discovery the visual areas of the net that own differences can be easily spotted by the developer. Again if simple changes have been made in the Petri net, such as the removal or the addition of net elements, the ImageNetDiff image will directly and clearly show the differences. If this is not the case and substantial changes have been made, at least the ImageNetDiff image points out the net areas which are of concern to the developer and which parts have not changed.

2.2 Technique

The tool makes use of the internal export function of Renew and the ImageMagick[3] tool kit. For the production of the differences image in the format of Portable Network Graphics (PNG) or alternatively Encapsulated Postscript (EPS) first the nets are being exported to the file system as image. Then the

[1] An alternative strategy is the avoidance of concurrent changes.

exported images are passed on as arguments to the imaging tool to compute the differences image, which will also be stored in the file system. The resulting image will feature light grayish drawing elements for the parts of the original images that are equal and two different shades of red for the additional and removed graphical parts Finally, for the convenience of the user the image is displayed by Renew once the computation of the differences image has finished. Sources of nets that are to be compared can be either nets that are opened within the editor of Renew or nets from the file system.

2.3 Constrains and Limitations

Several limitations to the presented method exist that result from the underlying tools. For a flawless comparison the compared images must have the same size. The comparison can not be customized, yet. For instance, the color scheme is fix. The results for nets in which all graphical elements have been moved are not satisfying, yet, because the images are compared coordinate pixel against coordinate pixel. There is no integration with the Petri net representation, yet. Thus, the discovery of changes is supported but the knowledge has to be transferred to the Petri net.

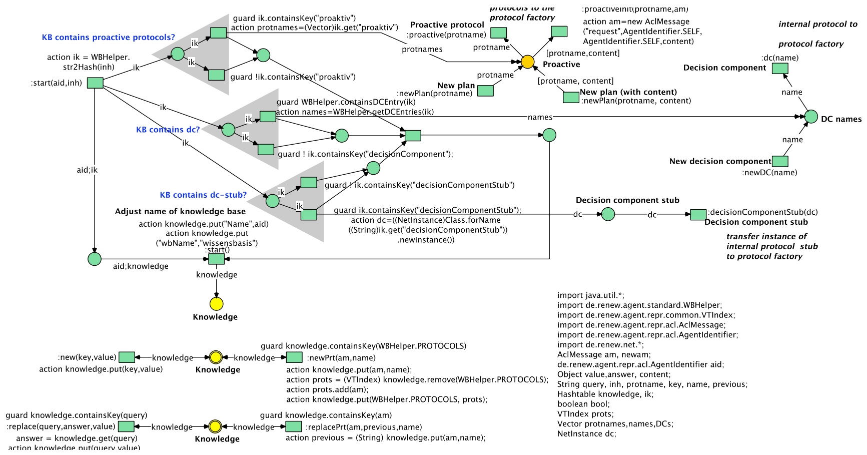
3 Example

As an example net for the presentation of the method we present the knowledge base net of the Mulan standard agents. The two nets differ – pragmatically – in the fact that they support two different property files formats: simple properties (*kb*) and XML notation (*kbe*, kb enhanced). The net that supports the enhanced representation is built upon the simple version, thus they are comparable. To find the similarities and differences of the implementation we present fragments of both nets in Figures 1 and 2. The fragments show the initialization of the net with the initial knowledge parts of the agents interface to the knowledge base and the interface that handles the initialization of decision components (active knowledge). Figure 3 then shows a screenshot of the resulting difference image (similar fragment).[2]

The developer's awareness is instantly attracted by the bright red net elements and inscriptions. One can see simple additions – manually marked in the image by dotted outlined squares – and also changes to the code / inscriptions – manually marked by dotted outlined ellipses –that have been made. The image shows clearly that all of the old net structure has been preserved. Only additional net elements and inscriptions have been added and some inscriptions have been altered.

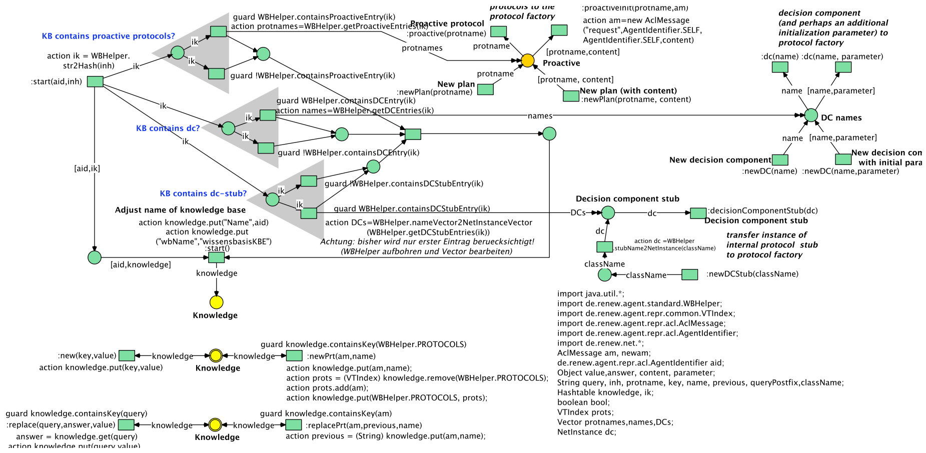
In a scenario of a shared development, if a developer is confronted by a concurrent change of the net, which results in a conflicting version of the net code, the tool can help the developer to decide whether the code has been manipulated,

[2] The dashed squares and ellipses are added manually.


KB contains proactive protocols?
guard ik.containsKey("proaktiv")
action protnames=(Vector)ik.get("proaktiv")
action ik = WBHelper.
str2Hash(inh)
:start(aid,inh)
guard !ik.containsKey("proaktiv")
Proactive protocol
:proactive(protname)
protnames
protname
:proactiveInit(protname,am)
action am=new AclMessage
("request",AgentIdentifier.SELF,
AgentIdentifier.SELF,content)
[protname,content]
Proactive
New plan
:newPlan(protname)
[protname, content]
New plan (with content)
:newPlan(protname, content)
internal protocol to
protocol factory
Decision component
:dc(name)
name
DC names
names
KB contains dc?
guard WBHelper.containsDCEntry(ik)
action names=WBHelper.getDCEntries(ik)
guard ! ik.containsKey("decisionComponent");
New decision component
:newDC(name)
aid;ik
KB contains dc-stub?
guard ! ik.containsKey("decisionComponentStub")
guard ik.containsKey("decisionComponentStub");
action dc=((NetInstance)Class.forName
((String)ik.get("decisionComponentStub"))
.newInstance())
Decision component stub
dc
:decisionComponentStub(dc)
transfer instance of
internal protocol stub
to protocol factory
Adjust name of knowledge base
action knowledge.put("Name",aid)
action knowledge.put
("wbName","wissensbasis")
:start()
aid;knowledge
knowledge
Knowledge
:new(key,value)
action knowledge.put(key,value)
guard knowledge.containsKey(WBHelper.PROTOCOLS)
:newPrt(am,name)
action knowledge.put(am,name);
action prots = (VTIndex) knowledge.remove(WBHelper.PROTOCOLS);
action prots.add(am);
action knowledge.put(WBHelper.PROTOCOLS, prots);
guard knowledge.containsKey(query)
:replace(query,answer,value)
answer = knowledge.get(query)
guard knowledge.containsKey(am)
:replacePrt(am,previous,name)
action previous = (String) knowledge.put(am,name);

```
import java.util.*;
import de.renew.agent.standard.WBHelper;
import de.renew.agent.repr.common.VTIndex;
import de.renew.agent.repr.acl.AclMessage;
import de.renew.agent.repr.acl.AgentIdentifier;
import de.renew.net.*;
AclMessage am, newam;
de.renew.agent.repr.acl.AgentIdentifier aid;
Object value,answer, content;
String query, inh, protname, key, name, previous;
Hashtable knowledge, ik;
boolean bool;
VTIndex prots;
Vector protnames,names,DCs;
NetInstance dc;
```

Figure 1. Knowledge base net template of a Mulan agent.


KB contains proactive protocols?
guard WBHelper.containsProactiveEntry(ik)
action protnames=WBHelper.getProactiveEntries(ik)
action ik = WBHelper.
str2Hash(inh)
:start(aid,inh)
guard !WBHelper.containsProactiveEntry(ik)
Proactive protocol
:proactive(protname)
protnames
protname
:proactiveInit(protname,am)
action am=new AclMessage
("request",AgentIdentifier.SELF,
AgentIdentifier.SELF,content)
[protname,content]
Proactive
New plan
:newPlan(protname)
[protname, content]
New plan (with content)
:newPlan(protname, content)
decision component
(and perhaps an additional
initialization parameter) to
protocol factory
:dc(name) :dc(name, parameter)
name [name,parameter]
DC names
names
KB contains dc?
guard WBHelper.containsDCEntry(ik)
action names=WBHelper.getDCEntries(ik)
guard !WBHelper.containsDCEntry(ik)
New decision component
:newDC(name) :newDC(name,parameter)
[aid,ik]
KB contains dc-stub?
guard !WBHelper.containsDCStubEntry(ik)
guard WBHelper.containsDCStubEntry(ik)
action DCs=WBHelper.nameVector2NetInstanceVector
(WBHelper.getDCStubEntries(ik))
Achtung: bisher wird nur erster Eintrag beruecksichtigt!
(WBHelper aufbohren und Vector bearbeiten)
Decision component stub
DCs
dc
:decisionComponentStub(dc)
action dc =WBHelper
.stubName2NetInstance(className)
className
:newDCStub(className)
transfer instance of
internal protocol stub
to protocol factory
Adjust name of knowledge base
action knowledge.put("Name",aid)
action knowledge.put
("wbName","wissensbasisKBE")
:start()
[aid,knowledge]
knowledge
Knowledge
:new(key,value)
action knowledge.put(key,value)
guard knowledge.containsKey(WBHelper.PROTOCOLS)
:newPrt(am,name)
action knowledge.put(am,name);
action prots = (VTIndex) knowledge.remove(WBHelper.PROTOCOLS);
action prots.add(am);
action knowledge.put(WBHelper.PROTOCOLS, prots);
guard knowledge.containsKey(query)
:replace(query,answer,value)
answer = knowledge.get(query)
guard knowledge.containsKey(am)
:replacePrt(am,previous,name)
action previous = (String) knowledge.put(am,name);

```
import java.util.*;
import de.renew.agent.standard.WBHelper;
import de.renew.agent.repr.common.VTIndex;
import de.renew.agent.repr.acl.AclMessage;
import de.renew.agent.repr.acl.AgentIdentifier;
import de.renew.net.*;
AclMessage am, newam;
de.renew.agent.repr.acl.AgentIdentifier aid;
Object value,answer, content, parameter;
String query, inh, protname, key, name, previous, queryPostfix,className;
Hashtable knowledge, ik;
boolean bool;
VTIndex prots;
Vector protnames,names,DCs;
NetInstance dc;
```

Figure 2. Enhanced knowledge base net template of a Mulan agent.

the syntax has not been changed and/or if the changes have been made in the same areas of the net. Thus, the manual act of merging the code or model can be significantly accelerated.

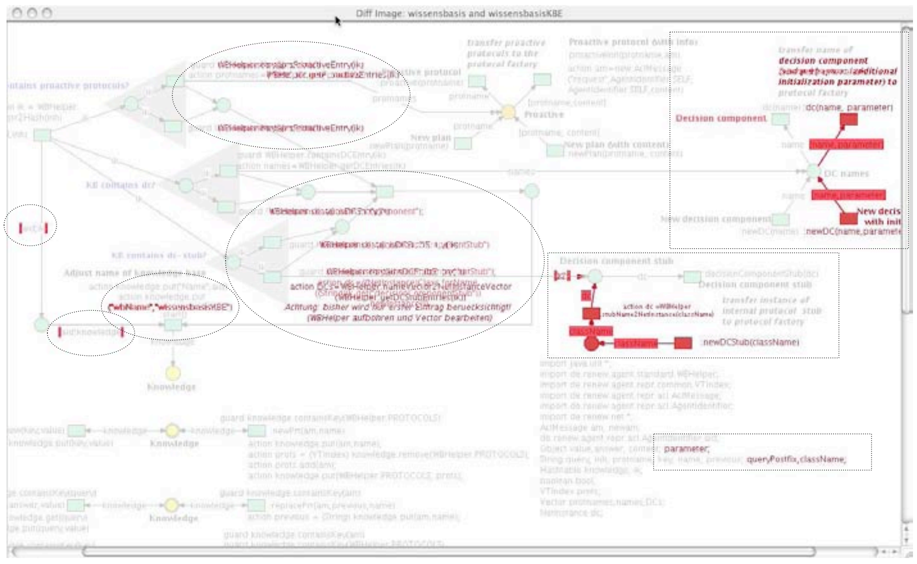


Figure 3. Screenshot showing differences of the two Petri nets.

4 Conclusion, Discussion and Outlook

Although the approach is rather simple, the results are effective and surprisingly efficient. Developers of Petri net models have the means to check for differences in their graphical code by the means of visual support. Clearly a code beautifier for Petri nets would improve the results of the ImageNetDiff plugin considerably. Here net components [1] could help to impose a conventionalized structure upon the nets.

The presented approach makes use of the graphical representation of the Petri nets, the export to an image format and the power of the graphical framework ImageMagick. There are, however, several other possibilities to tackle the presented problem. A similar possibility would be if the Petri net views in the editor could use layered canvas and alpha channels. Thus, one could easily find differences between versions, which are loaded in overlapping layers. One could compute equality of Petri nets on the ground of the formal representation including node and arc ids. Alternatively, one could program a diff tool on the ground of an exchange format such as PNML.

The presented method and the tool leaves room for many improvements. By choosing different color schemes for the diff images the readability could be improved significantly. However, since the used tools main purpose of comparing images is not concerned with graph representations, it does not support this feature and a reimplementation or switch to another tool could – with some effort – produce better results. The interpretation of the graphically highlighted elements could lead to a integration of useful information within the Petri net editor to further support the merging of concurrent changes.

Principally, with the presented method the results from image processing have to be re-transferred to the application domain. Alternatively, similar differences can be computed and presented to the developer on the direct analysis of Petri net structures. Here, additional information could support the process of matching elements in Petri net versions. For instance, id-tagged net elements (in Renew transitions and places have ids) could be matched. However, this would not solve the problem of constructs that have different ids but are syntactically equal. A method based on a Petri net representation is also less general than the presented method, which can be applied to other graphs such as UML diagrams.

In the future we want to investigate possibilities to automatically merge concurrent non-conflicting changes within a source code management system. Here, the questions of mergeable representations of graph structures and reimplementations of diff tools that may handle Petri net code are in the focus of research.

References

1. Lawrence Cabac, Michael Duvigneau, and Heiko Rölke. Net components revisited. In Daniel Moldt, editor, *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents. MOCA'06*, number FBI-HH-B-272/06, pages 87–102, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany, June 2006.
2. Gnu diff utilities. online, 2008. `http://www.gnu.org`.
3. Imagemagick homepage. online, 2008. `http://www.imagemagick.org/`.
4. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – the Reference Net Workshop. Available at: `http://www.renew.de/`, July 2008. Release 2.1.1.
5. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer, June 2004.

Fiona
A Tool to Analyze Interacting Open Nets

Peter Massuthe and Daniela Weinberg

Humboldt–Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{massuthe,weinberg}@informatik.hu-berlin.de

Abstract. Fiona is a tool that has been designed to check behavioral correctness of a service and to analyze the interaction of services in service oriented architectures, for instance. It implements very efficient data structures and algorithms, which have partly been adapted from LoLA. Fiona has been proven to be applicable in practice by service designers, service publishers, and service brokers. This tool paper describes the functionality of Fiona and provides an insight into its architecture.

1 Introduction

In this paper we present Fiona (available at `http://www.service-technology.org/fiona`), a tool to analyze the interaction of services. Its features cover the check of controllability and the construction of an operating guideline of a service as well as other derived notions. *Controllability* [1, 2] is a minimal correctness criterion of a service stating the existence of a behaviorally compatible partner for the service. An *operating guideline* (OG) [3, 4] of a service is an operational characterization of *all* behaviorally compatible partners of this service.

As a formal model for services we use *open nets* [5, 4], a special class of Petri nets that extend classical Petri nets by an interface for communication with other open nets. This idea is based on the module concept for Petri nets which was first proposed by Kindler [6].

The development of Fiona started in 2006 as a reimplementation of a tool called Wombat (available at `http://www.informatik.hu-berlin.de/top/wombat`). Wombat was designed to constructively decide controllability of an open net (called workflow module in Wombat). The main reasons for a reimplementation were (1) a completely new theoretical foundation for deciding controllability, (2) a desired focus on efficiency rather than mere effectiveness, (3) the newly developed concept of an operating guideline, and, thus, (4) the need for a separation of computing compatible interactions from reduction rules to ensure efficiency of the algorithms.

Currently, Fiona is developed distributedly by the groups of Wolfgang Reisig (Humboldt-Universität zu Berlin) and Karsten Wolf (University of Rostock, Germany). It is written in C++ and released as free software under the terms of the

GNU General Public License. Fiona's distribution is based on the GNU autotools, which provide the possibility to run Fiona on most operating systems.[1]

The functionality of Fiona comprises (among others) the following features:

Controllability. Controllability of an open net N is decided by synthesizing a partner of N (as an automaton called *interaction graph* (IG) [2]). If no partner can be synthesized (i.e. the IG is empty), then N is un-controllable.

Operating guideline. An operating guideline (OG) [4] is a finite characterization of *all* behaviorally compatible partners by annotating a single partner (as automaton) with Boolean formulae in order to derive all other partners.

Matching. Given an open net M and an operating guideline OG_N of an open net N, Fiona decides whether M is behaviorally compatible to N by matching M with OG_N. Matching is more efficient than composing the two nets and model checking the composition (as proposed by other approaches, like the *public view* approach).

Partner synthesis. Given an open net N, Fiona computes a behaviorally compatible partner open net M (if possible). The synthesis can be triggered to construct a small M (with respect to communication) or a partner M which exhaustively communicates with N.

Substitutability. Given two open nets N and N', Fiona can compare their sets of compatible partners using the corresponding OGs OG_N and $OG_{N'}$. If $OG_{N'}$ comprises OG_N, then N can be substituted by N' without rejecting any compatible partner. Different notions of substitutability are described in [7]. The corresponding decision procedures are implemented in Fiona.

Adapter generation. Given two open nets M and N which are *not* behaviorally compatible, we can consider their composition as an open net and ask Fiona to synthesize a partner. If such a partner exists, it constitutes an *adapter* A, mediating between M and N and making the composition of M, N, and A well-behaving by construction. For more details see [8].

Fiona is a stand-alone tool, designed to be used as a background service of existing service modeling tools. Therefore, Fiona has no graphical interface — the analysis task as well as the input file(s) are given to Fiona via command line options. Fiona then computes and reports the result and, if needed, generates the output file(s).

2 Context

Fiona can be used within the new computing paradigms of service-oriented computing (SOC) and service-oriented architectures (SOA) (see [9] for an overview), as well as other areas of intra- and interorganizational business process modeling and analysis [10]. Therein, a *service* represents a self-contained software unit that offers an encapsulated functionality via a well-defined interface. Services are used as building blocks to implement complex, highly dynamic, and flexible

[1] It compiles on MS Windows® (with Cygwin), Unix (Solaris), Linux, and Mac® OS.

business processes. SOAs introduce a *service broker* to organize the challenges of *service discovery*, i.e. the publishing and management of available services and the introduction of procedures to enable a client to find and use such a service.

Controllability is a minimal requirement for the correctness of a service and is particularly relevant for service designers. Operating guidelines are suited to support service discovery and can be used to decide substitutability of services or to generate adapters for incompatible services [8]. Thus, FIONA is intended to be used by service designers, by service providers, and by service brokers.

3 How to Use FIONA

FIONA accepts two different types of files as its input. The open net file format (`.net`) has been adapted from the LoLA [11] file format. It is fairly easy to read and to comprehend. So, it is possible to model an open net manually. Additionally, there exists a compiler BPEL2oWFN (available at `http://service-technology.org/bpel2owfn`) that implements a feature-complete open net semantics [12] for WS-BPEL and automatically generates an open net file for a given WS-BPEL process. Further, FIONA is able to read a textual representation of an operating guideline stored in a special file format (`.og`).

In the following we describe the main use cases of FIONA.

Controllability. Given an open net as a `.net` file, FIONA decides controllability by constructing an IG. The corresponding command is:

```
> fiona -t ig nets/example.net
```

After computing the IG, FIONA reports whether the net is controllable or not. Furthermore, a graphic file showing the IG is generated by invoking GraphViz Dot (available at `http://www.graphviz.org`).

Operating guideline. Given an open net as a `.net` file, FIONA constructs the OG of the net using the following command.

```
> fiona -t og nets/example.net
```

FIONA reports whether N is controllable or not and a graphic file showing the OG is generated. Further, a file called `example.net.og` is generated that represents the OG of the net textually.

Matching. If a `.net` file and an `.og` file is given, FIONA can decide whether the given net is behaviorally compatible to the net that the given OG corresponds to. The following command is used for invoking the matching algorithm.

```
> fiona -t match nets/client.net ogs/service.net.og
```

FIONA will report whether the net matches with the given OG or not. In case the matching fails, FIONA reports where the failure manifests in.

Partner synthesis. By passing a `.net` file to FIONA it is possible to construct a partner of that net. FIONA can construct two different types of partners: a very small one or a partner that exhaustively communicates with the net.

```
> fiona -t smallpartner nets/example.net
> fiona -t mostpermissivepartner nets/example.net
```

In both cases FIONA will create an open net `example-partner.net` that represents the partner. This is done with the help of the tool Petrify (available at `http://www.lsi.upc.es/~jordicf/petrify/distrib/home.html`).

FIONA provides some more analysis and construction features. A list of all features is available by invoking FIONA with the parameter `-help`. Using the parameter `-d 1`, ..., `-d 5` it is possible to retrieve detailed information of computation progress and debug information. Furthermore, the IG and OG graphics can be enriched to show more details of the graphs. Additionally, there exists a message bound parameter (`-m`) that refers to the *k-limited communication* [4] of open nets. The message bound limits the number of tokens that an interface place can carry simultaneously. The default is `-m 1`, i.e. interface places are safe.

4 Implementation Details

In the following we describe the main architecture of FIONA to construct an IG and an OG, which are the two basic features of FIONA and provide the basis of most other functionalities of FIONA.

In either case, an automaton called *communication graph* (CG) is constructed first, from which both IG and OG will be derived. The CG consists of *nodes* and *events*. The events reflect possible sending or receiving actions of a partner P of the considered net N. Each node q of the CG contains the set of markings of N, which can be reached by consuming and producing the messages along any path from the initial node of the CG to q. That set of markings is called *knowledge* at q ($k(q)$, for short) [4], representing the hypothesis of P about the state of N.

Representation of markings. The markings of N are stored in a data structure which was adapted from LoLA. The more markings it stores, the more the structure converges into a decision tree. It has proven to be very space efficient while allowing to decide the containment of a marking in linear time.

Knowledge calculation. The knowledge $k(q_0)$ of the root node q_0 of the CG consists of all markings reachable from the initial marking m_0 of N. For calculating the knowledge of $q \neq q_0$, $k(q)$ is initially filled with those markings derived from the predecessor nodes' knowledges that represent the occurrence of the event(s) leading to q. Then, FIONA computes the markings m' that are reachable from an $m \in k(q)$ and adds these m' to $k(q)$. The computation of the reachability graph is done as in LoLA. For instance, instead of backtracking, we fire a transition backwards.
The calculation of $k(q)$ can be enhanced by using stubborn sets. Here, only representative markings are stored in $k(q)$ (FIONA parameter `-R`) which may significantly reduce the space required for storing the CG.

Node classification. Each node is classified as either red or blue. The blue nodes constitute the IG/OG later on, whereas a red node represents a state of the partner P that is behaviorally incompatible with the open net N. Such nodes must be avoided and are not part of the IG/OG. Initially, every node q is blue. Then, we analyze $k(q)$ for non-final deadlocks and check for each such deadlock whether it is resolvable by an event of P. If some deadlock is not resolved, q is set to red. If $k(q)$ contains a marking that violates the message bound, q is set to red, too. By backtracking, the red color of q is propagated to its predecessors. To speed up the computation, red nodes are not deleted but stored to avoid a repeated computation of such a node.

Annotating a node. For each node q a Boolean annotation $\phi(q)$ (in conjunctive normal form) is stored. The annotation of q is uniquely defined by $k(q)$: each deadlock $d \in k(q)$ is represented by a clause c where each literal of c represents an event that resolves d. The annotation is part of the OG later on, but is also used to drive the order in which the successor nodes are computed (IG and OG).

$\phi(q)$ can also be used as an early classification of q: each literal x corresponds to an event leading from q to a node q' in the CG. If q' is red, then x is set to *false*. If thereby $\phi(q)$ becomes unsatisfiable, q becomes red immediately.

Successor node computation. For each blue node q we perform each event that occurs in $\phi(q)$, i.e. each event resolving at least one deadlock. For being able to apply the early classification as often as possible, we sort the clauses of $\phi(q)$ by its length and follow events of short clauses first.

In case of IG computation, we only consider a subset of the possible events: several reduction rules [2] for the CG are proven to preserve controllability.

The data structure of the CG is the basis of deciding controllability and for constructing an OG. The OG provides the basis of the matching algorithm. The IG/OG is used to construct a partner (small/most permissive) for a given net.

Controllability. The IG contains the blue nodes of the CG which was computed by applying all reduction rules. The annotations and knowledge values of the nodes are discarded. The open net N is controllable if and only if the root node of the CG is blue. Fiona reports this decision (and statistical numbers of the IG and CG sizes) on the command line; a graphical representation of the IG is generated.

Operating guideline. The OG consists of the blue nodes of the CG, too, this time computed without reductions. Again, the knowledges are discarded, but the annotations remain. In the graphical representation of the OG the annotation of a node q is shown inside of q. Fiona generates a graphical and a textual representation of the OG and stores both in separate files.

Matching. The matching algorithm is a check whether (1) the given net M is simulated by the given operating guideline OG_N and (2) M satisfies all annotations of OG_N. Therefore, Fiona performs a coordinated depth-first search through the state space of M and OG_N and evaluates each formula $\phi(q)$ by interpreting the currently enabled transitions of M as an assignment for $\phi(q)$. Eventually, Fiona will report the matching result. In case that M does not match, Fiona will report the marking of M and the node of OG_N where the simulation or the annotation is violated.

Partner synthesis. Fiona can synthesize two types of partners for a given open net N. A small partner is constructed based on the IG of N, or a most-permissive partner is computed out of the OG of N. Either graph provides the basis of the input of the tool Petrify, which creates the corresponding open net M. M is behaviorally compatible with N by construction.

5 Conclusion

We have presented the tool Fiona that is designed to be used by service designers, service brokers, and service publishers. Some of its data structures and algorithms have been adapted from LoLA. Further, we have put great effort on the theoretical basis of our algorithms in order to make the computations efficient with respect to time and memory consumption. Our case studies show that Fiona is indeed suitable to be used in practice [4, 2].

Fiona has been integrated into the ProM framework, a process mining tool kit with plug-able architecture (available at `http://prom.sourceforge.net/`).

References

1. Schmidt, K.: Controllability of Open Workflow Nets. In: EMISA 2005. LNI, Bonner Köllen Verlag (2005) 236–249
2. Weinberg, D.: Efficient controllability analysis of open nets. In: WS-FM 2008. LNCS, Springer-Verlag (2008) accepted.
3. Massuthe, P., Schmidt, K.: Operating Guidelines – An Automata-Theoretic Foundation for Service-Oriented Architectures. In: QSIC 2005, Melbourne, Australia, IEEE Computer Society Press (2005) 452–457
4. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: ICATPN 2007. Volume 4546 of LNCS., Springer-Verlag (2007) 321–341
5. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. AMCT **1**(3) (2005) 35–43
6. Kindler, E.: A compositional partial order semantics for Petri net components. In: ICATPN 1997. Volume 1248 of LNCS., Springer-Verlag (1997) 235–252
7. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding Substitutability of Services with Operating Guidelines. Technical Report 222, Humboldt-Universität zu Berlin, Germany (2008) accepted for a journal.
8. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Technical Report CS-02-08, Universität Rostock, Germany (2008) submitted to a conference.
9. Papazoglou, M.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (2007)
10. Aalst, W.M.P.v.d., Massuthe, P., Stahl, C., Wolf, K.: Multiparty Contracts: Agreeing and Implementing Interorganizational Processes. Technical Report 213, Humboldt-Universität zu Berlin, Germany (2007) submitted to a journal.
11. Schmidt, K.: LoLA: A Low Level Analyser. In: ICATPN 2000. Volume 1825 of LNCS., Springer-Verlag (2000) 465–474
12. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007. Volume 4937 of LNCS., Brisbane, Australia, Springer-Verlag (2008) 77–91

Autorenverzeichnis