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.

$A : (P \times T \times R) \to X$ 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.

$M_0 : P \to \Sigma_{MS}$ 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.

$L_w : (T \times P) \to \Sigma$ 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 $L_r : X \times \Sigma_{MS} \to \Sigma$. 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:

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

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