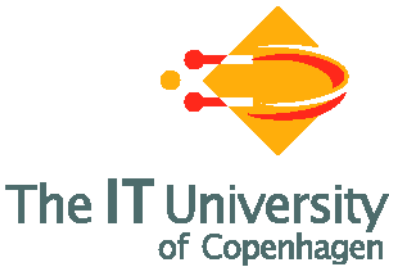

Business Process Execution with Bigraphs and Reactive XML

Thomas Hildebrandt
Henning Niss
Martin Olsen

IT University Technical Report Series **TR-2006-85**

ISSN 1600–6100 **April 2006**

ISSN 1600–6100

ISBN 87-7949-125-1

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web `www.itu.dk`

Business Process Execution with Bigraphs and Reactive XML

Thomas Hildebrandt, Henning Niss, and Martin Olsen*

IT University of Copenhagen {hilde,hniss,mol}@itu.dk

Abstract. Bigraphical Reactive Systems have been proposed as a meta model for global ubiquitous computing generalising process calculi for mobility such as the pi-calculus and the Mobile Ambients calculus as well as graphical models for concurrency such as Petri Nets. We investigate in this paper how Bigraphical Reactive Systems represented as Reactive XML can be used to provide a formal semantics as well as an extensible and mobile platform independent execution format for XML based business process and workflow description languages such as WS-BPEL and XPDL. We propose to extend the formalism with primitives for XPath evaluation and higher-order reaction rules to allow for a very direct and succinct semantics.

1 Introduction

Recently proposed language standards for business process coordination such as XPDL [8] and WS-BPEL [3] (combining the languages XLANG [33] and WSFL [21]) have a syntax based on XML to facilitate exchange of process descriptions between heterogeneous process execution engines and analysis tools. The semantics of the present business process execution languages, relating the process description to the possible state changes, is given by an informal specification.

Reliable implementations of business process execution engines and tools must be based on a formal semantics to be able to provide guarantees for the process execution and in particular to guarantee consistency between different process engines and tools.

Business processes are so-called *long-lived* processes and the state of running processes, sometimes referred to as *cases*, are continuously persisted. Not only mobile process descriptions but also mobile cases are highly relevant, e.g. if a business case is needed on a disconnected PDA, in a different part of the world or the process execution engine is updated and the cases must be exported to the updated engine. To facilitate mobile cases one would need a standard, platform independent representation of the *state* of processes as for instance an intermediate execution format like Java bytecode. Such a standard has not yet been defined, on the contrary, the state of a business process is usually assumed to be persisted in a proprietary format in a relational database [14].

In the present report we describe a general approach to define a platform independent execution format for business processes based on XML as well as an XML-based extensible format for the formal process language semantics derived from the meta process model of *Bigraphical Reactive Systems* (BRS) [11, 20, 24, 25] which can be seen as a specialized kind of graph rewriting systems. The BRS meta model prescribes a format for process language signatures and reaction rules (rewrite rules) used to describe the formal operational semantics, and a general theory for deriving from the reaction rules a labelled bisimulation congruence for the process language [11]. Processes are represented as two graphs (hence the name *bi*graphs), named the place graph and the link graph respectively, which are designed to generalise the pi-calculus [26] and the Mobile Ambients calculus [5]; bigraphs has been shown also to encompass Petri Nets [23].

* Authors listed alphabetically. This work was funded in part by the Danish Research Agency (grant no.: 2059-03-0031) and the IT University of Copenhagen (the Bigraphical Programming Languages project)

Concretely, we investigate how BRSs, by exploiting similarities between Bigraphs and XML, can be used to provide a formal semantics and execution format for XML-based business process languages, using a small subset of WS-BPEL as an illustration of the idea. In spite of being just a small subset, the WS-BPEL case provides several benefits. Firstly, the case illustrates how an industry standard XML-based programming language can be extended to an XML-based execution format using ideas from process calculi. Secondly, we show how the semantics can be given as XML-based rewrite rules thereby both providing an extensible and interchangeable format for the semantics and narrowing the gap usually arising between a programming language and its formalisation, as it is the case for π-calculus formalisations of business processes. Finally, the case suggests an interesting extension of BRS to allow for (linear) higher-order reaction rules and tree logics, in this concrete case a subset of XPath. The higher-order reaction rules is essentially a format for wide reaction rules in which the different parts of the rule may be nested inside each other, and thus parameters of the reaction rules may be contexts. Subsequently, we employ XPath to constrain such context parameters, resulting in a kind of *context-dependent* reaction rules.

Our formalisation is presented as an instance of a distributed meta process calculus *DiX*, which at the same time can be regarded as a term language for a (generalisation of) pure open bigraphs and a process calculus notation for tuples of (unordered) XML, XML contexts and XML-rewrite rules. The *DiX* calculus and the theory of bigraphical reactive systems form the theoretical foundation for a distributed XML-centric model of computation. This has been implemented in a prototype called Distributed Reactive XML; it provides an extensible, distributed (peer-to-peer) process execution engine for the calculus based directly on the formalisation, and in particular a process engine for any concrete instance. Our approach thus constitutes a general approach to developping extensible and distributed process execution engines from formal process semantics.

The *DiX* calculus with first-order reaction rules and its implementation as Distributed Reactive XML was presented in [18] based on the Reactive XML implementation given in [17, 36]. The work presented in this report builds on the MSc thesis by the 3rd author [27]. The case of WS-BPEL and and its bigraphical semantics using higher-order reaction rules is published in [19].

Related Work Much work has been carried out recently on formalisations of workflow languages, in particular within the Petri Net [29] and pi-calculus formalisms. Indeed the question of which of these two formalisms is most suitable has raised a lively debate [35]. The work in [31] describes a complete Petri Net-semantics for WS-BPEL. Pi-calculus formalisations of business and workflow processes have been described in [30, 32]. With respect to comprehensibility and extensibility, all of these formalisations suffer from the fact that the business process language is formalised in very abstract models with few primitives for interaction and reaction. In comparison, bigraphical reactive systems allow one to describe domain-specific reaction rules, just as in usual graph rewriting systems. Consequently, the process semantics given in the present paper stays very close to the WS-BPEL language by utilizing the extensibility of bigraphical reactive systems and the similarities between bigraphs and XML. An argument for employing abstract minimalistic models such as the π-calculus and Petri Nets is of course to be able to perform formal reasoning and utilize verification tools. We retain this hope by relying on the formal theory for bigraphical reactive systems, notably the theory of bisimulation congruences, which will be pursued in future work.

Our representation of bigraphs as XML is inspired by the similarities between process calculi for mobility and semi-structured data observed in [4] and is closely related to the work in [10]. However the focus of [10] is to *represent XML as bigraphs* (and using bigraph-logics introduced by the same authors in [9] to describe properties of XML) as opposed to the present paper that have the opposite focus, namely to *represent bigraphs as XML*, and using XML as an platform independent execution format for bigraphical reactive

systems. A possible joining point of the two lines of work would be to use bigraph-logics in place of the XPath constraints.

The work on XML-based execution formats relates to the proactive XML-centric models of computation and coordination surveyed in [7], in which processes that manipulates XML-documents are embedded in the documents themselves. In particular our work relates to the Workspaces approach [34] in which XPDL process descriptions are transformed into sets of XML documents representing the steps to be carried out, thus providing a distributed XML representation of the process state. The main difference between the Workspace approach and ours is that the computation steps in Workspaces are based on XSLT, which has the clear benefit of being an open and widely implemented standard. On the other hand, XSLT is in itself a complex programming language without a formal semantics.

Finally, it would be interesting to investigate the applicability of other graph rewriting models to give semantics of business process.

2 Bigraphical Reactive Systems and Reactive XML

In this section we describe the **Di**stributed e**X**tensible process (and context) calculus *DiX* (previously presented in [18]) and its relationship to bigraphs and representation as Reactive XML.

Notation: We let n, m, i, j range over natural numbers and write $[m]$ for the set (ordinal) $\{1, \ldots m\}$.

2.1 Signatures and Process expressions

The starting point for the extensible process calculus *DiX* is a general notion of signatures that encompasses both the signatures of XML documents and bigraph signatures. The terminology is partly borrowed from bigraph signatures.

Definition 1. *A* signature *is a tuple* $(\Sigma, \Xi \uplus \Delta \subseteq \Sigma, N_c \subseteq N, Att, ar)$*, where* Σ *is a set of* controls *ranged over by* κ*,* Ξ *and* Δ *are resp. the subsets of* active *and* atomic *controls,* N *is an infinite set of* names *ranged over by* n*,* N_c *is a set of* constant *names,* Att *is a set of finite* attribute index sets*, and* $ar : \Sigma \to Att$ *is a function assigning an attribute index set to each control.* □

As a signature for XML, Σ is a set of XML *element names*, N is a set of XML *attribute values and variables* where N_c is the subset of attribute values (concretely the strings *not* beginning with a \$), and Att is the set of finite sets of XML *attribute names*. The subset of *active* controls Ξ in the signature determines where *reactions* (or rewrites) can take place as described below and the subset of *atomic* controls Δ are controls that can not have any children. Following [10] we assume the existence of an atomic control with no attributes for each possible #PCDATA node.

Example 1 (Signature for WS-BPEL process descriptions). When viewed as *DiX* terms WS-BPEL processes are constructed from controls such as sequence, flow, while, and variable, each corresponding to a WS-BPEL element. The active controls allow reactions under the control; for example sequence and flow are active, hence allowing for the execution of the sub-processes. The passive controls does not allow such reactions; for example variable and while are passive since a variable declaration does not involve execution, and execution of a while body does not proceed until the condition has been met. Some controls carry attributes; for example variable carries an attribute $name$, $ar(\mathsf{variable}) = \{name\}$. In concrete processes attribute values are constants (in the set N_c) such as `amount`; in reaction rules attribute values are typically variables (in the set $N \backslash N_c$) such as `$x`. Refer to Figure 2 for the signature for the subset of WS-BPEL considered in this paper. □

The notion of constant names is an extension of the notion of bigraph signatures and will be explained when we introduce contexts below. For bigraph signatures the attribute index set Att is the set $\omega = \{[n] \mid n \geq 0\}$ of finite ordinals and the attribute indexes $ar(\kappa)$ of a bigraph control κ are referred to as the *ports* of κ. The attributes of bigraph controls are thus simply a list of names indicating which name each port is *linked* to. In other words, bigraph signatures has the form $(\Sigma, \Xi \uplus \Delta \subseteq \Sigma, \emptyset, \omega, ar)$.

A distributed Σ-process is an ordered set of unordered trees for which each node is labelled by a control $\kappa \in \Sigma$ and a set of name attributes indexed by $ar(\kappa)$, which we write as $\kappa\{a_i : n_i\}_{a_i \in ar(\kappa)}$. If $ar(\kappa) = \{a_1, \ldots, a_k\}$ the node $\kappa\{a_i : n_i\}_{a_i \in ar(\kappa)}$ corresponds to the XML element <κ a_1="n_1" $\cdots$ a_k="n_k">.

Definition 2. *For a signature $\Sigma = (\Sigma, \Xi \uplus \Delta \subseteq \Sigma, N_c \subseteq N, Att, ar)$ the Σ-processes are given by the grammar*

$$w ::= w \parallel w \mid p \mid 0 \qquad \text{wide } \Sigma\text{-processes}$$
$$p ::= \kappa\{a_i : n_i\}_{a_i \in ar(\kappa)}.p \mid \kappa_a\{i : n_i\}_{i \in ar(\kappa_a)}.1 \mid p \mid p \mid 1 \qquad \text{prime } \Sigma\text{-processes}$$

where $\kappa \in \Sigma \backslash \Delta$, $\kappa_a \in \Delta$ and $n_i \in N$. □

We use a commutative and associative binary parallel composition $\mid$ to separate siblings and the prefix notation $\kappa\{a_i : n_i\}_{a_i \in ar(\kappa)}.p$ for a tree with root $\kappa\{a_i : n_i\}_{a_i \in ar(\kappa)}$ and sub tree p. For XML the prefix operator corresponds to surrounding p with the usual open and close elements as in <κ a_1="n_1" $\cdots$ a_k="n_k"> p </κ>. We collect trees into an ordered set of trees by an associative binary parallel composition $\parallel$. Using bigraph terminology, we refer to $\mid$ as the *prime* parallel composition and $\parallel$ as the *wide* parallel composition. We also refer to trees as *prime* processes and ordered collections of trees as *wide* processes (rather than distributed processes). We let 0 denote the empty collection of trees (i.e. wide process) and 1 the emtpy tree (i.e. prime process).

Example 2 (WS-BPEL processes). In *DiX*, WS-BPEL processes are constructed from controls by prefixing (corresponding to the nesting in the XML rendition) and parallel composition (corresponding to juxtapositioning in the XML rendition). The following prime process expression copies in parallel the value from variable x to variable y and vice versa.

$$\begin{aligned}&\mathsf{flow}.\big(\mathsf{assign.copy}.(\mathsf{from}\{var : x\} \mid \mathsf{to}\{var : y\})\\ &\quad \mid \mathsf{assign.copy}.(\mathsf{from}\{var : y\} \mid \mathsf{to}\{var : x\})\big)\end{aligned}$$ □

We let $\equiv$ be the structural congruence defined as the least congruence with respect to the operators above that makes $\mid$ associative and commutative with identity 1 and $\parallel$ associative with identity 0.

Definition 3. *The structural congruence $\equiv$ is the least congruence on process expressions such that*

$$p_1 \mid (p_2 \mid p_3) \equiv (p_1 \mid p_2) \mid p_3 \qquad p \mid q \equiv q \mid p \qquad p \mid 1 \equiv p \qquad 1 \mid p \equiv p$$

and

$$r_1 \parallel (r_2 \parallel r_3) \equiv (r_1 \parallel r_2) \parallel r_3 \qquad r \parallel 0 \equiv r \qquad 0 \parallel r \equiv r$$ □

Associativity of the parallel compositions allows us to leave out parentheses, writing respectively $\prod_{i \in [n]} p_i$ and $\parallel\!\!\!\prod_{i \in [n]} p_i$ for the n times prime and wide parallel compositions; we let $\prod_{i \in \emptyset} p_i = 1$ and $\parallel\!\!\!\prod_{i \in \emptyset} p_i = 0$. As usual we will often leave out trailing nil processes, writing $\kappa_a\{a_i : n_i\}_{a_i \in ar(\kappa)}$ for $\kappa_a\{a_i : n_i\}_{a_i \in ar(\kappa)}.1$. We say that the *width* of a wide process expression $\parallel\!\!\!\prod_{i \in [n]} p_i$ is n, i.e. it is the *wide* parallel product of n primes.

2.2 Context expressions and reactions

To define reactions formally we first need to define *contexts* formally. Borrowing from bigraphs our contexts consist of two components W and σ, a *process context* and an *attribute context* respectively. The process context W is what one may first expect of a (multi-hole) context, namely a process expression with indexed holes $[\ ]_j$ in which processes can be placed. The attribute context σ is a finite name substitution that act as a context of the attribute variables. An attribute context allows renaming, fusion and instantiation of attribute variables. In bigraph terminology, the process context is called the *place graph* and the attribute context is called the *link map*.

Definition 4. *For a signature $\Sigma = (\Sigma, \Xi \uplus \Delta \subset \Sigma, N_c \subseteq N, Att, ar)$ the Σ-*contexts *are pairs $G = (W, \sigma)$, where $\sigma : N \to N$ is a finite substitution respecting constant names referred to as the* attribute context *and W is the* process context *defined by the grammar*

$$W ::= W \parallel W \mid P \mid 0$$
$$P ::= \kappa\{i : n_i\}_{i \in ar(\kappa)}.P \mid \kappa_a\{i : n_i\}_{i \in ar(\kappa_a)}.1 \mid P \mid P \mid 1 \mid [\ ]_j$$

where $\kappa \in \Sigma \backslash \Delta$, $\kappa_a \in \Delta$, $n_i \in N$, and $j \geq 0$. Define the names $n(W)$ of a process context W to be the set of names appearing as values of attributes in the expression W. □

That the substitution σ is finite means that the set $dom(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. That it respects contant names means that $dom(\sigma) \cap N_c = \emptyset$. We will say that an attribute context σ is *trivial* if it is the identity $id : N \to N$ (i.e. $dom(\sigma) = \emptyset$) and often identify W and (W, id).

Example 3 (WS-BPEL context). Let W be the process context

$$\mathsf{assign.copy.}(\mathsf{from}\{var : \$f\} \mid \mathsf{to}\{var : \$t\}) \mid [\ ]_1$$

then (W, id) is a context capturing an assignment from one variable (identified by the attribute variable $\$f$) to another ($\$t$) possibly in parallel with other processes ($[\ ]_1$). □

We type contexts $(W, \sigma) : (n, X) \longrightarrow (m, Y)$ if W has width m and for any hole $[\ ]_j$ in W the index j is in $[n]$, and the attribute context satisfies that $dom(\sigma) \subseteq X$ and $\sigma(X \cup n(W)) \subseteq Y$. Using bigraph terminology we refer to (n, X) and (m, Y) as *interfaces*, and (n, X) as the *innerface* and (m, Y) as the *outerface* of $(W, \sigma) : (n, X) \longrightarrow (m, Y)$.

Note that contexts are not uniquely typed: The innerface may contain names not appearing in the domain of σ and the outerface may contain names not appearing in $\sigma(X \cup n(W))$. For a typed context $(W, \sigma) : (n, X) \longrightarrow (m, Y)$ we write $(W, \sigma) \oplus X'$ for $(W, \sigma) : (n, X \uplus X') \longrightarrow (m, Y \cup X')$, the *extension* of the interfaces with a set of names X' satisfying $X' \cap X = \emptyset$. The condition ensures that $X' \cap dom(\sigma) = \emptyset$ and thus well-typedness.

Example 4 (Typed WS-BPEL context). A type of the context above is

$$(W, id) : (1, \{\$f, \$t\}) \longrightarrow (1, \{\$f, \$t\})$$

because it has one hole, uses the attribute values $\{\$f, \$t\}$ and has width 1. □

Example 5 (Extended typed WS-BPEL context). The typed context of Example 4 can be extended, for instance with two constant names x and y to $(W, id) \oplus \{x, y\}$ yielding the type

$$(W, id) : (1, \{\$f, \$t, x, y\}) \longrightarrow (1, \{\$f, \$t, x, y\})$$ □

We say that a context $(W, \sigma) : (n, X) \longrightarrow (m, Y)$ is *affine* if the same index appears at most once at a hole and that a context $(W, \sigma) : (n, X) \longrightarrow (m, Y)$ is *linear* if all indexes in $[n]$ appear exactly once. For bigraph signatures, the typed linear contexts given above is

a term language for open pure bigraphs [11]. That the bigraphs are open and pure means respectively that we do not have the usual constructor for local names used to represent name binding in the pi-calculus nor the possibility of binding names within the attributes of controls, as e.g. used for the input prefix in the pi-calculus. Local names and binding are allowed in general in binding bigraphs, but they are not needed for the work presented in this paper and is thus left for future work. We let 0 be short for the empty interface $(0, \emptyset)$. As usual, a process p can be viewed as a context $(p, id) : 0 \longrightarrow (m, Y)$ with trivial attribute context and the empty innerface, referred to as a *ground* context. We will often abbreviate the type of a ground context as $(p, id) : (m, Y)$.

Contexts compose by process substitution and attribute value substitution as defined formally below.

Definition 5. *For contexts $(W, \sigma) : (n, X) \longrightarrow (m, Y)$ and $(W', \sigma') : (m, Y) \longrightarrow (k, Z)$ define the composite context*

$$(W', \sigma') \circ (W, \sigma) = (W'(W\sigma'), \sigma' \circ \sigma) : (n, X) \longrightarrow (k, Z) \ ,$$

where $W\sigma'$ is the context obtained from W by substituting all attribute values n with $\sigma'(n)$ and $W'(W\sigma)$ is the context obtained from W' by for all indexes $i \in [m]$ inserting the i'th prime of $W\sigma$ into every i-indexed hole of W'. The composition of substitutions $\sigma' \circ \sigma$ is standard function composition.

The proposition below means that typed contexts and interfaces form a category.

Proposition 1. *Context composition is associative and $(\mathbb{I}\!\mathbb{I}_{i \in n}[\ ]_i, id) : (n, X) \to (n, X)$ is the identity context for the interface (n, X).*

Proof. Using associativity of process context substitution and function composition

$$\begin{aligned}
(W'', \sigma'') \circ \big((W', \sigma') \circ (W, \sigma)\big) &= (W'', \sigma'') \circ (W'(W\sigma'), \sigma' \circ \sigma) \\
&= \big(W''((W'(W\sigma'))\sigma''), \sigma'' \circ (\sigma' \circ \sigma)\big) \\
&= \big(W''((W'(W\sigma'))\sigma''), (\sigma'' \circ \sigma') \circ \sigma\big) \\
&= \big(W''(W'\sigma''((W\sigma')\sigma'')), (\sigma'' \circ \sigma') \circ \sigma\big) \\
&= \big(W''(W'\sigma''(W\sigma'' \circ \sigma')), (\sigma'' \circ \sigma') \circ \sigma\big) \\
&= \big(W''(W'\sigma'')(W\sigma'' \circ \sigma'), (\sigma'' \circ \sigma') \circ \sigma\big) \\
&= (W''(W'\sigma''), \sigma'' \circ \sigma') \circ (W, \sigma) \\
&= \big((W'', \sigma'') \circ (W', \sigma')\big) \circ (W, \sigma)
\end{aligned}$$

Example 6 (WS-BPEL context composition). Consider the context $(W, id) : (1, \{\$f, \$t\}) \longrightarrow (1, \{\$f, \$t\})$ from Example 4 above. From (W, id) we may obtain contexts representing each of the two assignments of the flow in Example 2 by compositions:

$$W_{x,y} = \mathsf{assign.copy}.(\mathsf{from}\{var : \mathtt{x}\} | \mathsf{to}\{var : \mathtt{y}\}) \mid [\,]_1 = ([\,]_1, [\$f \mapsto x, \$t \mapsto y]) \circ W$$

and

$$W_{y,x} = \mathsf{assign.copy}.(\mathsf{from}\{var : \mathtt{y}\} | \mathsf{to}\{var : \mathtt{x}\}) \mid [\,]_1 = ([\,]_1, [\$f \mapsto y, \$t \mapsto x]) \circ W$$

We can then obtain the complete process of Example 2 by combining these:

$$\begin{aligned}
&\mathsf{flow}.\big(\ \mathsf{assign.copy}.(\mathsf{from}\{var : x\} \mid \mathsf{to}\{var : y\}) \\
&\qquad \mid\ \mathsf{assign.copy}.(\mathsf{from}\{var : y\} \mid \mathsf{to}\{var : x\})\big) \\
&= \mathsf{flow}.[\,]_1 \circ W_{x,y} \circ W_{y,x} \circ 1
\end{aligned}$$

The contexts $W_{x,y}$, $W_{y,x}$, and $\mathsf{flow}.[\,]_1$ can be typed as follows:

$$\begin{aligned}
W_{x,y} &: (1, \emptyset) \longrightarrow (1, \{x, y\}) \\
W_{y,x} &: (1, \{x, y\}) \longrightarrow (1, \{x, y\}) \\
\mathsf{flow}.[\,]_1 &: (1, \{x, y\}) \longrightarrow (1, \{x, y\})
\end{aligned}$$

□

We say that a context is *active* if no holes are nested inside non-active controls. The dynamics of a process language is then defined by a set of parametric reaction rules.

Definition 6. *For a signature Σ define the set of* parametric reaction rules $PReact_\Sigma$ *as* $\{(W_L, W_R) | W_L : (n, X) \longrightarrow (m, Y), W_L$ *is linear*, $W_R : (n, X) \longrightarrow (m, Y)\}$.

For a set $R \subset PReact_\Sigma$, and contexts $W : (k, Z)$ and $W' : (k, Z)$ we say that $W \longrightarrow W'$ if there exists a parametric rule $\big(W_L, W_R : (n, X) \longrightarrow (m, Y)\big) \in R$, an active context $W_A : (m, Y \cup X') \longrightarrow (k, Z)$ and a parameter process $W_P : (n, X \uplus X')$ such that $W \equiv W_A \circ W_L \oplus X' \circ W_P$ and $W' \equiv W_A \circ W_R \oplus X' \circ W_P$. □

2.3 Reactive XML

We now turn to the correspondence between *DiX* and XML. In the following we let ϵ denote the empty document. As indicated above we represent controls as XML elements (except for the #PCDATA controls represented as character data) and attributes as XML-attributes. We use the reserved[1] element names `wide`, `reg`, and `hole` for respectively the root of the wide process, the root of the primes (referred to as regions in bigraphs) and the holes. The `hole` element has an attribute `name` providing the index of the hole. In summary, we define a translation $[\![\cdot]\!]$ from *DiX* processes and process contexts (that is, contexts with trivial attribute context) to XML as follows:

$$
\begin{array}{lll}
[\![\Vert_{i\in[n]} P_i]\!] & = \texttt{<wide>} & \\
 & \quad \texttt{<reg>}[\![P_1]\!]\texttt{</reg>} \ldots \texttt{<reg>}[\![P_n]\!]\texttt{</reg>} & \\
 & \texttt{</wide>} & \\
[\![\kappa\{a_i : x_i\}_{a_i \in ar(\kappa)}.P]\!] & = \texttt{<}\kappa\ a_1\texttt{="}x_1\texttt{"} \ldots a_n\texttt{="}x_n\texttt{">}\ [\![p]\!]\ \texttt{</}\kappa\texttt{>}, & \text{for } |ar(\kappa)| = n \\
[\![\kappa_a]\!] & = \kappa_a, & \text{for } \kappa_a \in \#\text{PCDATA} \\
[\![P \mid P']\!] & = [\![P]\!][\![P']\!] & \\
[\![1]\!] & = \epsilon & \\
[\![[\ ]_j]\!] & = \texttt{<hole name="j"/>} &
\end{array}
$$

Example 7. The process in Example 2 is rendered in XML as:

```
<wide>
  <reg>
    <flow>
      <assign>
        <copy>
          <from>42</from>
          <to variable="x"/>
        </copy>
      </assign>
      <assign>
        <copy>
          <from>Hello World</from>
          <to variable="y"/>
        </copy>
      </assign>
    </flow>
  </reg>
</wide>
```

[1] Technically, this can be reserved using the notion of XML namespaces.

The context in Example 4 is rendered in XML as:

```
<wide>
  <reg>
    <assign>
      <copy>
        <from var="$f"/>
        <to   var="$t"/>
      </copy>
    </assign>
    <hole name="1"/>
  </reg>
</wide>
```

□

In the implementation, described in Sec. 4, we represent the set of reaction rules as an XML document containing the rules encoded as pairs of contexts as well as an XPath representation of the active controls as will be described in Sec. 3.

3 Formalising XML Business Process Execution

When representing WS-BPEL processes as bigraphs we leverage the fact that Reactive XML provides an XML-based syntax for bigraphs and that a bigraphical reactive system may tailor the exact expressions to the application. In other words, the representation (almost) makes it possible to view the original WS-BPEL process expression as a Reactive XML expression.

In this section we investigate how to formalise XML business process execution, concretely a hybrid of BPEL4WS 1.1 and WS-BPEL 2.0, as bigraphical reactive systems. The contributions of this are twofold: on the one hand it gives a succint representation of the semantics of a WS-BPEL subset, on the other hand it directly provides a subsequent implementation based on our earlier work on Reactive XML [18] as described in Sec. 4.

For a bigraphical reactive system, one gets to specify not only process expressions in the formalism, but also the reaction rules. This makes bigraphical reactive systems particularly well-suited for representing the semantics of WS-BPEL as we can capture the semantics of each kind of WS-BPEL process as one or more bigraphical reaction rules.

3.1 A subset of WS-BPEL as processes

Figure 1 gives the grammar of the WS-BPEL process language we consider presented in the more compact *DiX* notation. We use $value$ to range over $\#$PCDATA and $expr$ to range over simple XPath expressions to be defined below. The translation to XML (as described in Section 2.3) is straightforward.

The corresponding signature Σ is defined in Figure 2. The signature mostly consists of controls corresponding directly to elements in WS-BPEL 2.0 and/or BPEL4WS 1.1 (those not marked with a star). The signature has additional controls (marked with a star) next, from_expr, and instance introduced by the representation and described below.

We employ a simple kind of *sorting* (i.e. schema) restricting the allowed children of controls and the allowed names for attributes. We let EXPR range over a subset of XPath expressions (including the contants $true$ and $false$), defined below. We use sets in sorts to represent disjunction and let $\&$ represent conjunction. We use $?$ for zero or one, and $*$ for zero or more. The process control thus have zero or one variables control as child and zero or one control from the set ACT (of actions).

$$
\begin{array}{lll}
\mathit{system} & ::= & \mathit{procs} \mid \mathit{state} \\
\mathit{procs} & ::= & \mathit{proc} \mid \ldots \mid \mathit{proc} \\
\mathit{state} & ::= & \mathit{inst} \mid \ldots \mid \mathit{inst} \\
\mathit{proc} & ::= & \mathsf{process}\{\mathit{name} : n\}.(\mathit{vars} \mid \mathit{act}) \\
\mathit{vars} & ::= & \mathsf{variables}.(\mathit{var} \mid \ldots \mid \mathit{var}) \\
\mathit{var} & ::= & \mathsf{variable}\{\mathit{name} : n\} \\
\mathit{act} & ::= & \mathit{seq} \mid \mathit{flow} \mid \mathit{while} \mid \mathit{if} \mid \mathit{inv} \mid \mathit{rec} \mid \mathit{rep} \mid \mathit{assign} \mid 1 \\
\mathit{seq} & ::= & \mathsf{sequence}.(\mathit{act} \mid \mathsf{next}.\mathit{act}) \\
\mathit{flow} & ::= & \mathsf{flow}.(\mathit{act} \mid \ldots \mid \mathit{act}) \\
\mathit{while} & ::= & \mathsf{while}.(\mathsf{condition}.\mathit{expr} \mid \mathit{act}) \\
\mathit{if} & ::= & \mathsf{if}.(\mathsf{condition}.\mathit{expr} \mid \mathsf{then}.\mathit{act} \mid \mathsf{else}.\mathit{act}) \\
\mathit{inv} & ::= & \mathsf{invoke}\{\mathit{operation} : n, \mathit{inputVariable} : n\} \\
\mathit{rec} & ::= & \mathsf{receive}\{\mathit{operation} : n, \mathit{variable} : n\} \\
\mathit{rep} & ::= & \mathsf{reply}\{\mathit{operation} : n, \mathit{variable} : n\} \\
\mathit{assign} & ::= & \mathsf{assign.copy}.(\mathit{from} \mid \mathit{to}) \\
\mathit{from} & ::= & \mathsf{from}\{\mathit{var} : n\} \mid \mathsf{from\_expr}.\mathit{expr} \\
\mathit{to} & ::= & \mathsf{to}\{\mathit{var} : n\} \\
\mathit{inst} & ::= & \mathsf{instance}\{\mathit{name} : n\}.(\mathit{instvars} \mid \mathit{act}) \\
\mathit{instvars} & ::= & \mathsf{variables}.(\mathit{instvar} \mid \ldots \mid \mathit{instvar}) \\
\mathit{instvar} & ::= & \mathsf{variable}\{\mathit{name} : n\}.\mathit{value}
\end{array}
$$

Fig. 1. Grammar for the WS-BPEL subset. Metavariables n ranges over names N, expr ranges over valid XPath expressions, and value ranges over $\#\text{PCDATA}$.

Control	Activity	Attributes	Sort
process	passive	{name:n}	{variables}$^?$&ACT$^?$
variables	passive		{variable}*
variable	passive	{name:n}	#PCDATA
sequence	active		ACT$^?$&next
next*	passive		ACT$^?$
flow	active		ACT*
while	passive		{condition}&ACT$^?$
if	passive		{condition}&{then}&{else}
condition	passive		EXPR
then	passive		ACT$^?$
else	passive		ACT$^?$
invoke	atomic	{operation:n, inputVariable:n}	$\emptyset$
receive	atomic	{operation:n, variable:n}	$\emptyset$
reply	atomic	{operation:n, variable:n}	$\emptyset$
assign	passive		{copy}
copy	passive		{from, from_expr}&{to}
to	atomic	{var:n}	$\emptyset$
from	atomic	{var:n}	$\emptyset$
from_expr *	passive		EXPR
instance*	active	{name:n}	{variables}$^?$&ACT$^?$

Fig. 2. WS-BPEL process signature. Let ACT be the set {sequence, flow, while, if, invoke, receive, reply, assign}, #PCDATA be the set of #PCDATA controls and EXPR be the subset of #PCDATA controls for which the value is a valid XPath expression. Controls marked with a * are introduced by the representation; the rest correponds to WS-BPEL elements.

3.2 Process instances and execution state

From a high-level perspective, a WS-BPEL process description consists of a number of processes in parallel

$$proc_1 \mid \ldots \mid proc_n$$

During execution, each of the processes $proc_i$ may get instantiated, eg., when it is being invoked. A process instance needs to maintain the current "program counter" indicating what activity is currently being executed and an assignment of values to the variables of the process. We shall refer to the representation of program counters and variable assignments for all process instances as the *execution state* of the WS-BPEL process description.

Traditionally, execution engines store execution state in proprietary formats, typically in a database. We propose to represent not only the WS-BPEL process description as XML, but also the execution state. This allows us to use Reactive XML to implement the execution steps taken by WS-BPEL processes. Reaction rules implement the semantics of WS-BPEL by rewriting the execution state appropriately. Again from a high-level perspective the current state of the execution of a WS-BPEL process description has the following form, represented by $system$ in the grammar:

$$(proc_1 \mid \ldots \mid proc_n) \mid (inst_1 \mid \ldots \mid inst_m)$$

In other words, it is a set of process descriptions together with a set of the currently instantiated processes. We need the descriptions in order to be able to instantiate new processes; the instances capture the execution state, not as program program pointers but, in the style of process calculi, as descriptions of the current state and possible future behaviour. Process instances are represented using the control instance which is just like process except variables carry a current value. Thus a typical instance has the form

$$\begin{array}{l}\mathsf{instance}\{name : \mathtt{i}\}.\\ \quad(\mathsf{variables}.(\mathsf{variable}\{name : x_1\}.v_1 \mid \cdots \mid \mathsf{variable}\{name : x_n\}.v_n)\\ \quad \mid p)\end{array}$$

where p represents the future behavior of the instantiated process.

Since process descriptions are only meant to be used when instantiating processes the process control is passive; dually, process instances are meant to be executed (ie., rewritten) and therefore the instance control is active.

For each syntactic construct we present a number of reaction rules specifying how execution of the construct proceeds. For example, there are three rules specifying how to execute conditionals. The reaction rules rewrite the XML representation of the execution state; specifically, the process instance for which an execution step is to be taken. Once execution of an activity has finished it is represented by the nil process 1. These reaction rules "capture" the semantics of WS-BPEL.

3.3 Activity composition

WS-BPEL defines a number of *structural activities* which combine smaller activities into a combined activity.

One of the most basic structural activities in WS-BPEL is that of *parallel* (or *concurrent*) composition, known as flow. Activities prefixed by a flow are executed in parallel/concurrently[2]. The execution of the flow activity ends when all parallel activities have finshed executing. By making the corresponding control flow active we ensure, appealing to the underlying bigraphical model, that the activities may execute in parallel. It would also

[2] Future work will address the representation of *links* to constrain the execution order.

have been possible to omit the explicit control completely, however, at the cost of more differences between the original WS-BPEL process and its encoding. A single reaction rule removes the flow control when all activities have ended:

$$\mathsf{flow}.1 \to 1 \tag{1}$$

An equally important structural activity is *sequential* composition through the sequence control. The activities are to be executed in the order in which they occur as children of the `<sequence>` element (ie., so-called "document order"). The execution of the sequence activity ends when the last activity in the sequence has finished executing. In contrast to flow its encoding has to address the fact that the children of a control are *unordered* in bigraphs. This means that we cannot just group two sequential activities under a sequence control which is active, as that would allow either of them to execute. Instead we introduce a new, passive control, next, to block execution of the second activity, and provide an explicit reaction rule for transferring control to the second activity once the first has finished. That is, we represent two WS-BPEL activities in sequence, `<sequence>` $act_1\ act_2$ `</sequence>`, by the process $\mathsf{sequence}.(p_1 \mid \mathsf{next}.p_2)$ (when p_i is the representation of act_i) and use the following reaction rule to transfer control:

$$\mathsf{sequence}.\mathsf{next}.[\,]_1 \to [\,]_1. \tag{2}$$

The if structural activity provides for the conditional execution of an activity. An activity is executed depending on the evaluation of a conditional expression (by default specified in XPath 1.0). To support basing conditions on XPath expressions we extend Reactive XML with a primitive, $EvalXPath(\cdot)$, for evaluating XPath expressions (rather than extending the underlying calculus, we could instead have written an XPath interpreter in *DiX*). Consider a rule containing $EvalXPath(\mathit{expr})$ on the right-hand side. Reactive XML rewrites using this rule by evaluating the XPath expression *expr* against the *DiX* context matching the left-hand side of the rule and inserting the result in place of $EvalXPath(\mathit{expr})$.

Equipped with this primitive we can easily specify the semantics of if by first appealing to the primitive (3) for computing the condition and then proceeding based on whether the condition evaluates to true (4) or false (5).

$$\begin{aligned}
&\mathsf{if}.(\mathsf{condition}.[\,]_1 \mid \mathsf{then}.[\,]_2 \mid \mathsf{else}.[\,]_3) && (3)\\
&\quad\to \mathsf{if}.(\mathsf{condition}.EvalXPath([\,]_1) \mid \mathsf{then}.[\,]_2 \mid \mathsf{else}.[\,]_3)\\
&\mathsf{if}.(\mathsf{condition}.\mathit{true} \mid \mathsf{then}.[\,]_1 \mid \mathsf{else}.[\,]_2) && (4)\\
&\quad\to [\,]_1\\
&\mathsf{if}.(\mathsf{condition}.\mathit{false} \mid \mathsf{then}.[\,]_1 \mid \mathsf{else}.[\,]_2) && (5)\\
&\quad\to [\,]_2
\end{aligned}$$

The allowed XPath expressions are boolean and simply-typed (i.e. integer) expressions over constants, and the functions `bpws:getVariableData('`n`')` for extracting the value of a variable (as in BPEL4WS 1.1).

The while structural activity provides for the repeated execution of an activity controlled again by an XPath expression. The activity is executed repeatedly until the XPath condition no longer evalutes to true, in which case the execution of the while activity ends.

We specify the semantics of while by (as usual) unfolding the while loop to an if construct (6) and then proceeding using the rules for if.

$$\begin{aligned}
&\mathsf{while}.(\mathsf{condition}.[\,]_1 \mid [\,]_2)\\
&\quad\to \mathsf{if}.\big(\mathsf{condition}.[\,]_1 \mid \mathsf{then}.(\mathsf{sequence}.([\,]_2 \mid \mathsf{next}.\mathsf{while}.(\mathsf{condition}.[\,]_1 \mid [\,]_2))) && (6)\\
&\qquad\quad \mid \mathsf{else}.1\big)
\end{aligned}$$

3.4 Variables

Assigning values to variables is one of the *primitive activities* of WS-BPEL (in the subset we consider the only other primitive activities are concerned with invoking processes as discussed in Section 3.7). Variable assignments take the form $\mathsf{assign.copy}.(\mathit{from} \mid \mathit{to})$.

The intention is to assign the value specified by from to the variable specified by to. In our WS-BPEL subset to can only specify a variable as in $\mathsf{to}\{\mathit{var} : \mathtt{x}\}$. The value to assign to the to variable is specified by from: it can be either another variable, $\mathsf{from}\{\mathit{var} : \mathtt{x}\}$, or an XPath expression, $\mathsf{from\_expr}.\mathit{expr}$. Below we first describe how to define variable assignments of the form $\mathsf{from}\{\mathit{var} : \mathtt{x}\}$; the form $\mathsf{from\_expr}.\mathit{expr}$ is simpler as it appeals simply to $\mathit{EvalXPath}(\mathit{expr})$ rather than involving looking up the current binding of a variable.

Recall, that process instances record the current bindings of values to variables, as in

$$\begin{array}{l}\mathsf{instance}\{\mathit{name} : \mathtt{assignex}\}.\\ \quad(\mathsf{variables}.(\mathsf{variable}\{\mathit{name} : \mathtt{x}\}.\mathit{17} \mid \mathsf{variable}\{\mathit{name} : \mathtt{y}\}.\mathit{Hello\ World})\\ \quad \mid \ldots)\end{array} \tag{7}$$

where the #PCDATA values *17* and *Hello World* are bound to x and y respectively.

Executing an assignment $\mathsf{assign.copy}.(\mathit{from} \mid \mathit{to})$ is therefore a matter of manipulating the correct variables in the instance's variables control. In order to not let an assignment from one process instance affect the variables of another instance, we need to insist that the controls assign and variables are both located under the same instance control, i.e. they are in the same scope. Furthermore, since the assignment may occur within a structural activity, we expect the reaction rule for variable-to-variable assignment to take the form:

$$\begin{array}{l}\mathsf{instance}\{\mathit{name} : \$i\}.(C(\mathsf{assign.copy}.(\mathsf{from}\{\mathit{var} : \$f\} \mid \mathsf{to}\{\mathit{var} : \$t\}))\\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{\mathit{name} : \$f\}.[\,]_1 \mid \mathsf{variable}\{\mathit{name} : \$t\}.[\,]_2 \mid [\,]_3))\\ \longrightarrow \mathsf{instance}\{\mathit{name} : \$i\}.(C(1)\\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{\mathit{name} : \$f\}.[\,]_1 \mid \mathsf{variable}\{\mathit{name} : \$t\}.[\,]_1 \mid [\,]_3))\end{array} \tag{8}$$

(where $[\,]_1$ is the value of the variable matched by $\$f$, $[\,]_2$ is the value of the variable matched by $\$t$, and $[\,]_3$ are the remaining variable bindings).

Intuitively, the context C above captures the fact that assign may be nested under active controls, i.e. flow, while, or a sequence. For example, considering again the process instance in (7) we could have

$$\begin{array}{l}\mathsf{instance}\{\mathit{name} : \mathtt{assignex}\}.(\mathsf{variables}.(\ldots)\\ \quad \mid \mathsf{sequence}.(\mathsf{assign.copy}.(\mathsf{from}\{\mathit{var} : \mathtt{x}\} \mid \mathsf{to}\{\mathit{var} : \mathtt{y}\}) \mid \mathsf{next}.\ldots))\end{array} \tag{7'}$$

in which case C therefore is $\mathsf{sequence}.([\,]_1 \mid \mathsf{next}.(\ldots))$.

Formally, we want to have an infinite set of rules obtained by instatiating C with all possible active contexts. In the next sections we will suggest a format of higher-order parametric reaction rules that allow us to specify such rule sets.

Analogously, we should like the rule for assigning results of XPath evaluations to variables to take the form:

$$\begin{array}{l}\mathsf{instance}\{\mathit{name} : \$i\}.(C(\mathsf{assign.copy}.(\mathsf{from\_expr}.[\,]_1 \mid \mathsf{to}\{\mathit{var} : \$t\}))\\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{\mathit{name} : \$t\}.[\,]_2 \mid [\,]_3))\\ \longrightarrow \mathsf{instance}\{\mathit{name} : \$i\}.(C(1)\\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{\mathit{name} : \$t\}.\mathit{EvalXPath}([\,]_1) \mid [\,]_3))\end{array} \tag{9}$$

Remark The notion of *closed* links present in the existing theory of *binding* bigraphs offer an alternative to the solution based on higher-order reaction rules that we propose. Intuitively, closed links correspond in XML to (unique) identifiers/keys and references to such. Instead of letting the *instance node* determine the scope one could then let variable names be identifers/keys and the uses of of variables references to such. We leave the exploration of closed links and its representation in XML for future work.

3.5 Higher-order Reaction Rules

Consider again the reaction rule for assignment. We wish to be able to abstract the process context C in the reaction rule by a hole, writing:

$$\begin{array}{l} \mathsf{instance}\{name : \$i\}.([\ \mathsf{assign}.(\mathsf{copy}.(\mathsf{from}\{var : \$f\} \mid \mathsf{to}\{var : \$t\}))\ ]_4 \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$f\}.[\ ]_1 \mid \mathsf{variable}\{name : \$t\}.[\ ]_2 \mid [\ ]_3)) \\ \longrightarrow \mathsf{instance}\{name : \$i\}.([\ 1\ ]_4 \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$f\}.[\ ]_1 \mid \mathsf{variable}\{name : \$t\}.[\ ]_1 \mid [\ ]_3)) \end{array} \tag{10}$$

The parameters of holes number 1, 2 and 3 are as usual prime processes, that is contexts $P_i : (1, X)$, but the parameter of hole number 4 is a prime process context $C : (1, Z) \longrightarrow (1, Z)$ (where $Z = \{\$f, \$t\}$) with a single hole. That is, we wish to instantiate (10) with a wide process context $W = P_1 \parallel P_2 \parallel P_3 \parallel C$ resulting in the ground rule

$$\begin{array}{l} \mathsf{instance}\{name : \$i\}.(C \circ (\mathsf{assign}.(\mathsf{copy}.(\mathsf{from}\{var : \$f\} \mid \mathsf{to}\{var : \$t\}))) \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$f\}.P_1 \mid \mathsf{variable}\{name : \$t\}.P_2 \mid P_3)) \\ \longrightarrow \mathsf{instance}\{name : \$i\}.(C \circ 1 \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$f\}.P_1 \mid \mathsf{variable}\{name : \$t\}.P_1 \mid P_3)) \end{array}$$

However, note that the parameter $W = P_1 \parallel P_2 \parallel P_3 \parallel C$ above has type $(1, Z) \longrightarrow (4, X \cup Z)$. Essentially, we need the hole of the process context C to be part of the *outerface* and the process inside the hole $[\ (\mathsf{assign}.(\mathsf{copy}.(\mathsf{from}\{var : \$f\} \mid \mathsf{to}\{var : \$t\})))\ ]_4$ to be part of the *innerface* of the redex.

To be able to make the contexts appearing in higher-order holes part of the innerface, we extend the types for interfaces with a limited linear function space considering interfaces of the form $t ::= (\bar{t}, X)$, where $\bar{t}$ is a vector of types $t_1 t_2 \ldots t_n$ which is short for $\otimes_i t_i \multimap 1$. A higher-order context W where the hole with index i is of the form $[\ W_i\ ]_i$ and W_i is a process of type t_i will then have the interface $(t_1 t_2 \ldots t_n, X)$. This means that a prime process context C of type $t_i \longrightarrow (1, X)$ can be placed in the hole $[\ W_i\ ]_i$, replacing the hole with the process $C \circ W_i$. We will write 0 for the empty interface $(\epsilon, \emptyset)$ where ϵ is the empty vector. As before, a prime process P can be regarded as a ground process context with type $P : 0 \longrightarrow (1, X)$. In particular, a higher order context hole $[\ 0\ ]_i$ then correspond to a normal (first order) hole $[\ ]_i$ in which a prime process P can be inserted.

The type of the redex and reactum in the reaction rule above is then $(\bar{t}, Z) \longrightarrow (1, Z')$ where $t_1 = t_2 = t_3 = 0$, $t_4 = (1, Z)$, $Z = \{\$f, \$t\}$, and $Z' = \{\$f, \$t, \$i\}$.

We write (n, X) for the type $(\bar{t}, X)$ where $|\bar{t}| = n$ and $t_i = 0$ for all $i \in [n]$, that is, the case where all holes are normal first order holes. This makes the higher-order contexts and interfaces a conservative extension of the first order contexts and interfaces.

To be able to make holes of process contexts part of the outerface we define the *involution* of a higher-order process context W to be the process $W^- : 0 \longrightarrow (\bar{t}, X)$ for $\bar{t} = t_1 t_2 \ldots t_n$ if $W \equiv \text{Ⅲ}_{i \in [n]} P_i$ and P_i can be typed $t_i \longrightarrow (1, X)$.

Now for $W = P_1 \parallel P_2 \parallel P_3 \parallel C$ we have $W^- : 0 \longrightarrow (\bar{t}, X \cup Z)$ for $t_1 = t_2 = t_3 = 0$ and $t_4 = (1, Z)$ matching the innerface $(\bar{t}, Z)$ of the redex and reactum (if it is extended with the names in $X \backslash Z$).

For the present paper we restrict ourself to only consider higher-order contexts of type $(\bar{t}, X) \longrightarrow (n, Y)$ and $0 \longrightarrow (\bar{t}, X)$ as given by the grammar below.

Definition 7. *For a signature $\Sigma = (\Sigma, \Xi, N_c \subseteq N, Att, ar)$ the* higher-order Σ-contexts *H are defined by the grammar*

$$\begin{array}{lll} H & ::= (W_{ho}, \sigma) \mid {W_{ho}}^- & \\ W_{ho} & ::= W_{ho} \parallel W_{ho} \mid P_{ho} \mid 0 & \\ P_{ho} & ::= \kappa\{i : n_i\}_{i \in ar(\kappa)}.P_{ho} \mid P_{ho} \mid P_{ho} \mid 1 \mid [\ W_{ho}\ ]_j & \end{array}$$

where $\sigma : N \to N$ are finite substitutions, $\kappa \in \Sigma$, $n_i \in N$, and $j \geq 0$ as for first-order contexts.

As indicated above, we type contexts $(W_{ho}, \sigma) : (\bar{t}, X) \longrightarrow (m, Y)$ for $\bar{t} = t_1 t_2 \ldots t_n$ if W_{ho} has width m, $dom(\sigma) \subseteq X$ and $\sigma(X \cup n(W_{ho})) \subseteq Y$, and for any hole $[\ W'_{ho}\ ]_j$ ${W'_{ho}}^-$ can be typed $0 \longrightarrow t_j$. We type contexts ${W_{ho}}^- : 0 \longrightarrow (\bar{t}, X)$ for $\bar{t} = t_1 t_2 \ldots t_n$ if $W_{ho} \equiv \Pi_{i \in [n]} P_i$ and P_i can be typed $t_i \longrightarrow (1, X)$.

We define higher-order context composition (inductively) as follows.

Definition 8. *For contexts $(W_{ho}, \sigma) : (\bar{t}, X) \longrightarrow (m, Y)$ and ${W'_{ho}}^- : 0 \longrightarrow (\bar{t}, X)$ define the composite context*

$$(W'_{ho}, \sigma) \circ W_{ho} = W'_{ho}[(P_i \circ {W''_{ho}}^-)\sigma / [\ W''_{ho}\ ]_i] : 0 \longrightarrow (m, Y)$$

for $W_{ho} \equiv \Pi_{i \in [n]} P_i$ and where $[(P_i \circ {W''_{ho}}^-)\sigma / [\ W''_{ho}\ ]_i]$ is the substitution of $(P_i \circ {W''_{ho}}^-)\sigma$ for holes $[\ W''_{ho}\ ]_i$ in W'_{ho}.

The higher-order contexts allow us to specify the reaction rule for assign as in (10) above. However, we wish to constrain the parameter of the fourth hole to only *active* contexts. In general the constraints may depend on attribute values, for instance to guarantee the existence of a certain path of controls between the root and the hole(s) as it is the case for the XPath addressing of sub contents of variables allowed in WS-BPEL. In the following section we address how this can be achieved.

3.6 XPath attribute values and context constraints

We consider a small subset of XPath given by the grammar

$$\begin{array}{lll} \phi & ::= & naos \mid expr \\ naos & ::= & \texttt{//*[not(ancestor-or-self::*[}nameset\texttt{])]} \mid \texttt{//*} \\ nameset & ::= & \texttt{name()='}n\texttt{'} \mid \texttt{name()='}n\texttt{'}\ \texttt{or}\ nameset \\ expr & ::= & \texttt{bpws:getVariableData('}n\texttt{')} \mid \ldots \end{array}$$

The XPath expressions defined by $naos$ are of the form

$$\texttt{//*[not(ancestor-or-self::*[name()='}n_1\texttt{' or ... or name()='}n_k\texttt{'])]}$$

and selects nodes *not* nested within any of the controls n_i for $i \in [k]$. These expressions are for instance used to identify active contexts, by letting the set $\{n_1, \ldots, n_k\}$ be the set of passive controls. We will let ϕ_{active} denote this expression. The XPath expressions defined by $expr$ are as for the while conditions, booleans and simple typed expressions.

Recall that an XPath expression is evaluated with respect to a node (somewhat confusingly referred to as the context) in an XML-document and results in a nodeset. We define that a prime context P *satisfies* an XPath constraint if all of the holes are children of one of the nodes in the nodesets resulting from evaluating XPath on the children of the reserved reg control of $[\![P]\!]$ (the context represented as XML). The syntax of higher-order context holes is then extended to $[\ W'_{ho}\ ]^{\phi}_j$, where ϕ is an XPath expression belonging to the subset defined above. We extend the interface types accordingly to $t ::= (\bar{t}, \bar{\phi}, X)$ where $\bar{t}$ as before is a vector $t_1 \ldots t_n$ of types and $\bar{\phi}$ is a vector $\phi_1 \ldots \phi_n$ of limited XPath expressions as defined by the grammar above. We omit the XPath constraints from the type if they all are the expression //* that selects all contexts.

We extend the typing condition to require for $(W_{ho}, \sigma) : (\bar{t}, \bar{\phi}, X) \longrightarrow (m, Y)$ for $\bar{t} = t_1 t_2 \ldots t_n$ that for any hole $[\ W'_{ho}\ ]^{\phi}_j$ $\phi = \phi_j$ and for the involuted contexts with XPath constraints ${W_{ho}}^- : 0 \longrightarrow (\bar{t}, \bar{\phi}, X)$ for $\bar{t} = t_1 t_2 \ldots t_n$and $\bar{\phi} = \phi_1 \ldots \phi_n$ if$W_{ho} \equiv \Pi_{i \in [n]} P_i$ and P_i can be typed $t_i \longrightarrow (1, X)$ and satisfies ϕ_i.

Returning to the assign case, we can now add the constraint ϕ_{active} to the hole with index 4 and type the redex (and reactum) $W_L : (\bar{t}, \bar{\phi}, X) \longrightarrow (1, X)$ where $t_1 = t_2 = t_3 = 0$ and $t_4 = (1, Z)$ and $\phi_1 = \phi_2 = \phi_3 =$ //* and $\phi_4 = \phi_{active}$.

3.7 Process communication

Communication amongst processes is the other form of *basic activities* of WS-BPEL we consider. The specification of communication takes up a large fraction of the WS-BPEL specification; here we shall focus on the basics of invoking a process and process communication. WS-BPEL addresses orchestration of web services and as such integrate features from WSDL (Web Services Description Language) [6]. In the present work, rather than working with web services, we consider a system as a collection of processes and interpret process invocation and communication as between the processes in the system. Furthermore, WS-BPEL also specifies how to correlate the messages between multiple (instances of) processes using so-called "correlation sets". See [27] for the details of representing this in Reactive XML.

A business process may invoke another process, thereby creating an instance of the invoked process, using $\mathsf{invoke}\{operation : \mathtt{op}, \ldots\}$. This creates an instance of the process in the system which contains a $\mathsf{receive}\{operation : \mathtt{op}, \ldots\}$ activity. The invoking process instance may specify parameters to the receiving process by including a variable in the invoke attribute $inputVariable$. The intention is to look up the current value of the variable in the instance, and bind that value to the formal parameter specified in the receive's $variable$ attribute (just as was done for variable assignment).

The above informal description can be expressed in the following reaction rule:

$$
\begin{array}{l}
\quad \mathsf{instance}\{name : \$i\}.([\ \mathsf{invoke}\{operation : \$o, inputVariable : \$in\}\ ]_3^{\phi_{active}} \\
\qquad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$in\}.[\,]_1 \mid [\,]_2)) \\
\mid\ \mathsf{process}\{name : \$p\}.([\mathsf{receive}\{operation : \$o, variable : \$var, \}]_6^{\phi_{active}} \\
\qquad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$var\} \mid [\,]_4) \mid [\,]_5) \\
\longrightarrow \\
\quad \mathsf{instance}\{name : \$i\}.([1]_3^{\phi_{active}} \mid \mathsf{variables}.(\mathsf{variable}\{name : \$in\}.[\,]_1 \mid [\,]_2)) \\
\mid\ \mathsf{instance}\{name : \$p\}.(\mathsf{variables}.(\mathsf{variable}\{name : \$var\}.[\,]_1 \mid [\,]_4) \mid [\,]_5) \\
\mid\ \mathsf{process}\{name : \$p\}.([\mathsf{receive}\{operation : \$o, variable : \$var\}]_6^{\phi_{active}} \\
\qquad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$var\} \mid [\,]_4) \mid [\,]_5)
\end{array}
$$

Observe (1) how the invoking process instance simply discards the invoke (which means it is asynchronous in the sense of BPEL since it does not assume a reply), (2) that the receiving process description remains unchanged (making it possible to create more instances), and (3) a new process instance has been added to the system with the correct variable binding and the "body" of the receiving process description ($[\,]_5$). We have used the same trick as for assign in order to locate the invoke under seq, flow, and while. One similarly needs a reaction that allows sending messages between two process instances (in WS-BPEL using reply and receive) following the pattern above:

$$
\begin{array}{l}
\quad \mathsf{instance}\{name : \$rp\}.([\mathsf{reply}\{operation : \$o, variable : \$out\}]_3^{\phi_{active}} \\
\qquad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$out\}.[\,]_1 \mid [\,]_2)) \\
\mid\ \mathsf{instance}\{name : \$rv\}.([\mathsf{receive}\{operation : \$o, variable : \$var\}]_6^{\phi_{active}} \\
\qquad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$var\}.[\,]_4 \mid [\,]_5)) \\
\longrightarrow \\
\quad \mathsf{instance}\{name : \$rp\}.([1]_3^{\phi_{active}} \mid \mathsf{variables}.(\mathsf{variable}\{name : \$out\}.[\,]_1 \mid [\,]_2)) \\
\mid\ \mathsf{instance}\{name : \$rv\}.([1]_6^{\phi_{active}} \mid \mathsf{variables}.(\mathsf{variable}\{name : \$var\}.[\,]_1 \mid [\,]_5))
\end{array}
$$

4 Implementing Business Process Execution with Reactive XML

In this section we describe the implementation of Distributed Reactive XML and its perspectives for business process execution and simulation. The implementation is based on

XML Store [2, 15, 28] as a peer-to-peer persistent storage layer. Our implementation extends the previous implementation presented in [18] by adding support for wide and higher-order reaction rules. The implementation and the examples are available on the web: ⟨`http://www.itu.dk/research/theory/bpl/reactivexml/`⟩.

4.1 XML Store

XML Store [2, 15, 28] is a general-purpose, peer-to-peer distributed, persistent storage manager for tree-structured data (XML documents). Below we briefly describe these features.

XML Store is a *storage manager* for tree structured values (data). XML Store provides functionality for storing and retrieving tree-shaped values—concretely, XML documents. Values are stored persistently, and as such outlives the application storing them. Once stored, a value is identified by a location-independent identifier (typically, a cryptographic hash of the contents of the value).

XML Store is *peer-to-peer distributed*. This means that an XML Store provides wide-scale distribution of the values it is storing. Distribution in XML Store is transparent so an application cannot observe whether a value is stored locally or remotely. Therefore, an application behaves identically whether values are distributed or not. XML Store can be based on any so-called structured peer-to-peer routing protocol; the current implementation is based on Kademlia [22].

XML Store is *value-oriented*. This means that data, once stored, does not change; in other words, data is *immutable*. This is analogously to the notion of *values* in programming languages (for example non-references in Standard ML, strings in Java, etc). The crucial idea in making XML Store value-oriented is that since values are never updated or changed, they can be cached, replicated, etc freely without the need for coherency protocols.

For example, the current state of the execution of a process is a value—it is never updated. Therefore we can freely cache it at (copy it to) all interested parties. On the other hand, we also have to take special measures to perform the equivalent of updates on the execution state. In XML Store this is a two-step process: first compute the new state by constructing a value representing the new state, then bind a name to a (unique) identifier for the value. For example, we might bind the name `state` to the identifier of the process expression. After one step of reductions, the name `state` gets bound to the (new) identifier for the new process expression. Such destructive updates are simple (they only involve a name and a 128-bit identifier) and occur only in isolated places when "updating" the current state. We shall refer to such updatable entities as *cells*. In pratice, the only form of update cells support are compare-and-swap operations: for a cell c we can update it to contain a new location-independent identifier if we know the identifier already contained in the cell. Refer to [16] for a justification of choosing compare-and-swap as the basic operation.

XML Store employs *sharing* aggressively. This means that rather than storing the same value (data item) multiple times, XML Store simply points to the same, already stored item. XML Store uses an asynchronous background process [2] that discovers shared values, discards all but one and updates pointers to the discarded values to point to the one remaining value. In this way, XML Store really stores DAGs rather than trees.

This avoids the obvious inefficiency in the example above: rather than constructing a completely new value for the new state, one reuses as much as possible the old value. For example, if a reaction takes place in the left child of node that has many more untouched (by the reaction) children, then when constructing the new value one simply reuse the untouched children (by using the pointers) to them. This is made possible since XML Store is value-oriented—in other words, it is guaranteed that noone changes the shared values.

4.2 Processes as XML in XML Store

As shown in Section 2.3 process expressions are mapped to XML in a very direct way.

Example 8. A process instance with two assignments in parallel in both *DiX* and XML syntax.

DiX:

```
instance{name : assignex}.
 (variables.variable{name : x}
 | flow.
    (assign.copy.
      (from.42 | to{var : x})
    | assign.copy.
      (from.What is the meaning?
      | to{var : x})
    )
 )
```

XML:

```
<instance name="assignex">
 <variables>
   <variable name="x"/>
 </variables>
 <flow>
  <assign>
   <copy>
     <from>42</from>
     <to variable="x"/>
   </copy>
  </assign>
  <assign>
   <copy>
     <from>
      What is the meaning?
     </from>
     <to variable="x"/>
   </copy>
  </assign>
 </flow>
</instance>
```

□

Architecturally, Distributed Reactive XML is an XML Store distributed over a number of peers, which provides clients with access to the current business process. Clients connect to this XML Store either by joining the peer-to-peer network, or as traditional clients. Since one could imagine different situation where each of them would be an advantage, it makes sense to have both options. For instance, a back-end ERP system which updates the processes on a regular basis would most likely benefit from being a part of the network, instead of connecting to the XML Store each time an update takes place. On the other hand, mobile clients or clients with less resources, for instance mobile PDAs, may not have resources available to join a peer-to-peer network, and they would therefore connect to the XML Store as clients.

The XML document making up the business process is distributed as well. Distribution is achieved by means of a peer-to-peer routing algorithm (for locating stored data) and each peer may store zero or more parts of the complete business process. Therefore, one typically finds that one peer stores one instance, say, another peer stores another instance, and so on. The distribution makes it possible to store an instance close to the peer for which the instance is currently relevant. For example, if the state of the overall system is

```
<system>
  <process name="pn_1"> p_1 </process>
  ...
  <process name="pn_n"> p_n </process>
  <instance name="in_1"> i_1 </process>
  ...
  <instance name="in_m"> i_m </process>
</system>
```

then when instantiating process pn_i the system evolves to

```
<system>
  <process name="pn_1"> p_1 </process>
  ...
  <process name="pn_n"> p_n </process>
```

```
  <instance name="in_1"> i_1 </process>
  ...
  <instance name="in_m"> i_m </process>
  <instance name="in_{m+1}"> i_{m+1} </process>
</system>
```

In this case it is natural, and indeed the current behavior, to store the new instance in_{m+1} at the peer that instantiated pn_i since it presumably needs to execute the instance. Should it later be the case that a different peer is responsible for most of the execution of the instance, or should the instantiating peer be subject to high load, then it is possible to move the corresponding part of the XML document to a new peer precisely because it is a value and hence will not be updated (albeit, the current prototype engine has no means to support this).

In order to keep track of the current state of the system, we maintain an updatable cell containing a value reference to the current state of the *entire* system. In other words, through the cell one gets access to the `<system>` element and all its children. When the state evolves the cell gets updated to point to the new state.

Note that the current prototype implementation of XML Store does not support *distributed* cells. Thus regardless of the distribution scheme chosen, we still need to update the cell to the overall state of the system *centrally*.

4.3 Implementing reaction rules

To implement the execution of reaction rules we need to be able to match left hand sides of parametric reaction rules and replace the result with the right hand side.

Recall that a *DiX*process p may react to be become a process p', $p \longrightarrow p'$, if there exists a reaction rule (W_L, W_R), a parameter W_P, and an active context W_A such that $p \equiv W_A \circ W_L \circ W_P$; we shall refer to $W_L \circ W_P$ as the *redex* (and $W_R \circ W_R$ as the *reactum*) and W_A as the *evaluation context* of the reaction. (Consult Definition 6 for the complete details.)

Let us first consider how to implement prime reaction rules, that is, reaction rules $(W_L : (n, X) \longrightarrow (1, Y), W_R : (n, X) \longrightarrow (1, Y))$. For a prime reaction rule we only need to consider evaluation contexts with one hole, where the redex is inserted. The redex is composed of W_L and W_P where W_P has width n, ie. $W_P = \amalg_{i \in [n]} p_i$. The composition is realized by inserting the i'th prime of W_P into the i'th hole of W_L. In other words, p can react using the rule (W_L, W_R) if

$$p \equiv W_A[W_L[i : P_i]] \tag{11}$$

(where $W[i : P]$ denotes filing all holes with index i of W with P) in which case

$$p' \equiv W_A[W_R[i : P_i]] \tag{12}$$

Therefore performing a reaction $p \longrightarrow p'$ amounts to finding a reaction rule such that we can solve equation (11), and then computing the result (12).

To solve equation (11) we must traverse p looking for $W_L[i : P_i]$ under active controls only (since W_A is required to be active). In the current prototype implementation we do so by constructing the set $\mathcal{P}_A$ of all sub-processes of p (p itself included) which are located only under active controls. Having constructed this set we look for elements of $\mathcal{P}_A$ (ie., processes) "matching" W_L; a process matches W_L if it can be obtained from W_L by inserting prime processes (even the nil process 1) in place of the holes of W_L. In other words, we match the "structure" of W_L. If a match is found, a reaction is possible.

Concretely, we compute the set $\mathcal{P}_A$ by an XPath expression. Let $xpath(\phi, [\![p]\!])$ denote the set of roots of subtrees in $[\![p]\!]$ that satisfies ϕ. From a signature $\Sigma = (\Sigma, \Xi \uplus \Delta \subset$

$\Sigma, N_c \subseteq N, Att, ar)$ we can construct an XPath expression that picks out only active contexts, we do so by disallowing all passive controls[3]:

$$\phi_{active}(\Sigma) = \texttt{//*[not(ancestor-or-self::*}$$
$$\texttt{[name()='}\kappa_1\texttt{' or name()='}\kappa_2\texttt{' or ... name()='}\kappa_k\texttt{'])]}$$

for $\Sigma \setminus \Xi = \{\kappa_1, \ldots, \kappa_n\}$. Now, $\mathcal{P}_A = xpath(\phi_{active}(\Sigma), [\![p]\!])$.

Example 9. For the WS-BPEL signature in Figure 2 the XPath $\phi_{active}(\Sigma)$ is

```
//*[not(ancestor-or-self::*[
      name()='process' or name()='assign' or
      name()='variables' or name()='variable' or
      name()='copy' or name()='from' or name()='to' or
      name()='to_query' or name()='from_query' or
      name()='from_expr' or name()='from_var' or
      name()='if' or name()='then' or name()='else' or
      name()='condition' or  name()='while' or
      name()='next' or name()='receive' or
      name()='invoke' or name()='reply'])]
```

□

As an example of the rewriting process, consider the rule for assignments of expressions to variables:

$$\begin{array}{l} \mathsf{instance}\{name : \$i\}.([\,\mathsf{assign.copy}.(\mathsf{from\_expr}.[\,]_1 \mid \mathsf{to}\{var : \$t\})\,]_4 \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$t\}.[\,]_2 \mid [\,]_3)) \\ \longrightarrow \mathsf{instance}\{name : \$i\}.([\,1\,]_4 \\ \quad \mid \mathsf{variables}.(\mathsf{variable}\{name : \$t\}.\mathit{EvalXPath}([\,]_1) \mid [\,]_3)) \end{array} \tag{13}$$

and an example process (repeated from Example 8)

$$\begin{array}{l} \mathsf{instance}\{name : \texttt{assignex}\}. \\ \quad (\mathsf{variables.variable}\{name : x\} \\ \quad \mid \mathsf{flow}. \\ \qquad (\mathsf{assign.copy}.(\mathsf{from}.42 \mid \mathsf{to}\{var : x\}) \\ \qquad \mid \mathsf{assign.copy}.(\mathsf{from}.\texttt{What is the meaning?} \mid \mathsf{to}\{var : x\}))) \end{array}$$

Performing a reaction on this process using the rule (13) then proceeds as follows:

1. *Find all posssible redexes by finding all evaluation contexts.*
   For the example process, the XPath expression (from Example 9) will identify the two assignment subexpressions.
2. *Match each of the possible redexes against the left hand side of the reaction rule instantiating holes and variables, possibly recursively matching within holes.*
   Matching the second assignment expression in the process against the left hand side of the reaction rule above will result in a match between the left hand side of the reaction rule (1) with the variable $\$iname$ instantiated to `assignex`, (2) with the context hole $[\,]_4$ bound to the context

   $$\mathsf{flow}.(\mathsf{assign.copy}.(\mathsf{from}.42 \mid \mathsf{to}\{var : x\}) \mid [\,]),$$

   (3) inside the context hole the matching is executed recursively, resulting in a match with (a) the rule hole $[\,]_1$ bound to `What is the meaning?` and (b) the variable $\$t$ bound to `x`, and (4) the rule holes $[\,]_2$ and $[\,]_3$ bound to the empty tree, i.e. the nil process 1, reflecting that there is no content of the x variable and no other variables.

[3] This is an implementation detail. Should we have specified that only active controls were present on the path to the document root, we would have had to also include the reserved elements `<wide>` and `<reg>` in the XPath expression.

3. *If any match exists, the reaction can be executed by calculating a reactum based on the right hand side of the reaction rule, and reconstructing the process expression.*
   Since all data stored in XML Store is immutable, clients cannot simply change the matched node (the redex) in the process tree to reflect the changes. Instead they have to build up a new tree. Figure 3 illustrates this situation. Before the reaction, the process is as seen in Figure 3(a). After the reaction, Figure 3(b), a new process has been built, but new nodes have only been constructed from the nodes which have to be "updated" (the reactum) up to the root. On the path to the root unchanged nodes are reused.

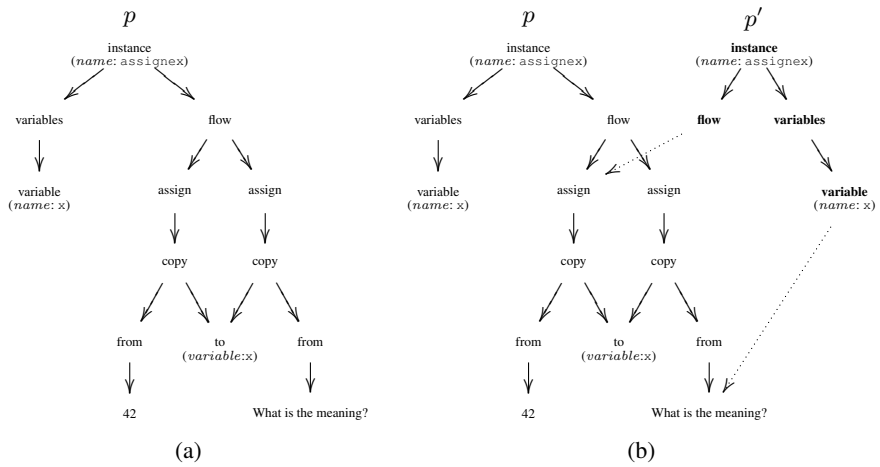

Fig. 3. A reaction $p \longrightarrow p'$; unchanged nodes are reused (dotted arrows indicate reuse; bold text indicate the newly constructed paths).

   The handle to the current process will at this point still refer to the old root node p. To make other clients aware of the new process, the client has to updated the handle to the new root p'.
   Such updates of handles (the only updates possible with XML Store) are done using an atomic compare-and-swap algorithm, which guarantees that nobody has changed the value in the time $\Delta t = [t_{read}; t_{swap}]$. By using this facility, we are able to obtain a simple distribution of client updates to the process. Thus ultimately, this is how coordination is implemented.

In the implementation, reaction rules are saved as a so-called rewrite rulesets. Concretely the rules are placed in a document with root element `REWRITE_RULESET` and two attributes, a `NAME` attribute and a `CONSTRAINT` attribute. The first simply provides a name to the ruleset used in the tool, the second provides the XPath expression determining the evaluation contexts for this ruleset (for example, the XPath given in Example 9).

Reaction rules are described within the ruleset as pairs of wide processes, respectively within a `RULE_LEFT` and a `RULE_RIGHT` tag.

Example 10. The reaction rule describing the semantics of assignment in the case of copying from an expression to a variable renders as follows.

```
<REWRITE_RULESET NAME="BPEL" CONSTRAINT=".....">

 <REWRITE_RULE NAME="copy from_expr - to" >
  <RULE_LEFT>
    <wide>
```

```
    <reg>
      <instance name="$iname">
        <CONTEXT_HOLE NAME="4">
          <assign>
            <copy>
              <from_expr>
                <RULE_HOLE NAME="1"/>
              </from_expr>
              <to var="$t"/>
            </copy>
          </assign>
        </CONTEXT_HOLE>
        <variables>
          <variable name="$t">
            <RULE_HOLE NAME="2"/>
          </variable>
          <RULE_HOLE NAME="3"/>
        </variables>
      </instance>
    </reg>
   </wide>
 </RULE_LEFT>
 <RULE_RIGHT>
   <wide>
    <reg>
      <instance name="$iname">
        <CONTEXT_HOLE NAME="4">
        </CONTEXT_HOLE>
        <variables>
          <variable name="$t">
            <EXEC_XPATH_HOLE NAME="4"/>
          </variable>
          <RULE_HOLE NAME="3"/>
        </variables>
      </instance>
 </RULE_RIGHT>
</REWRITE_RULE>

.....

</REWRITE_RULESET>
```
□

4.4 Synchronizing updates

The simple form of synchronization mentioned above works, but does not support situations where several clients simultaneously inspect the current process, find possible reactions, and build up a new process. To handle this, we will allow non-conflicting reactions (intuitively, reactions in *different* parts of the process) to take place concurrently. We use the term *conflicting reactions* to denote the situation where we are not able to incorporate changes from two (or more) reactions without leaving the process in an inconsistent state.

Assume that the two reaction rules $\mathcal{R}_1 = (L_1, R_1)$ and $\mathcal{R}_2 = (L_2, R_2)$ are performed on the same process. The reactions are performed simultaneously, consequently, they will inspect the process in the exact same state. We can now state two situations with conflicting reactions:

1. The two reactions overwrite each other's changes. Since they are both changing the same nodes, we cannot fuse the changes from both reactions to one process tree.
2. One (or both!) of the reactions makes changes to the redex for the other reaction. Since a reaction is only possible if the rule matches the redex, this situation removes the initial condition for one or both of the reactions.

As described in Section 4.3 performing a reaction on the process p, amounts to finding a matching subtree (a redex) t_L in p and replacing this with a new subtree (the corresponding reactum) t_R. Assume now that when performing $\mathcal{R}_1$, a subtree t_{L_1} in p is found. Additionally, a subtree t_{L_2} is found for $\mathcal{R}_2$ in p. We know that all nodes changed when performing $\mathcal{R}_1$ must be within the subtree t_{L_1}, and all nodes changed when performing $\mathcal{R}_2$ must be within the subtree t_{L_2}. Hence, a conservative estimate for non-conflicting reactions are: if $\mathcal{R}_1$ does not change any nodes in t_{L_2} and likewise $\mathcal{R}_2$ does not change any nodes in t_{L_1}, the two reactions will not have any overlapping changes.

Let $subtree$ be the function that for a node n returns a set containing all nodes in the the tree with root n.

Definition 9. *Consider two reaction rules $\mathcal{R}_1 = (L_1, R_1)$ and $\mathcal{R}_2 = (L_2, R_2)$, the redex t_{L_1} of the reaction $\mathcal{R}_1$ performed on p, and the redex t_{L_2} of the reaction $\mathcal{R}_2$ performed on p. The two reactions $\mathcal{R}_1$ and $\mathcal{R}_2$ are* conflicting*, if $subtree(t_{R_1}) \cap subtree(t_{R_2}) \neq \emptyset$*

We can use this knowledge in an optimistic concurrency control manager, where we allow clients to inspect the process expression at any time. The client will then find possible reactions. When it is ready to commit the result of one of these reactions, we *validate* whether the reaction is in conflict with other reactions performed in the time between the client inspected the process and the attempted commit operation. If any reactions occured, for each of them we check that the redex for that reaction does not have any nodes in common with the redex for the reaction we are about to commit. If there are no conflicts, we can incorporate the changes from this reaction in the shared process. In case of conflicts, we simply abort the commit operation.

In order to be able to do this validation, we need to track each reaction performed and the matching subtree (redex) that was the condition for the reaction. We capture these in so-called versions. A *version* consists of the *resulting* process tree and a changeset. A *changeset* records the changes that takes the original process tree (before the reaction took place) to the process tree stored in the version. Therefore, a changeset consists of the redex, the resulting reactum, and a XPath expression indicating what part of the process tree was rewritten.

Example 11. Consider again the reaction for executing the second assignment in the example above. In that case the version contains the process tree depicted in Figure 3(b) and a changeset. The changeset contains the redex (the tree in Figure 3(a))), the reactum (the tree labeled p' in Figure 3(b)), and an XPath expression indicating the path to the instance

```
/child::*[1]
```

□

We can now describe what is really stored in the XML Store, namely the latest version together with a list of versions leading to that version. The aggresive use of sharing in XML Store avoids the obvious problem of repeatedly storing the same (parts of) process trees again and again.

Note that above we also have a match with the first assignment. A concurrent reaction would produce a changeset with the same redex and XPath expression, but with a different reactum. Since the redices are the same, we have a conflict. As an example of reactions that *can* occur without conflicts, consider a number of clients performing (non-conflicting) reactions on the constiuents of a flow-control; hence the implementation indeed allows for the concurrent execution of processes in a flow. Another example is the "adminstrative reactions": for example removing the flow-control when all constiuents have finished (similarly for sequence), and evaluating XPath expressions.

As a side effect of storing changesets, we are able to track all changes on a reaction-by-reaction basis. This gives us a nice feature for debugging ReactiveXML.

5 Conclusion and Future Work

We have demonstrated how Bigraphical Reactive Systems, by exploiting the similarities of Bigraphs and XML, can be used to provide a formal semantics and a mobile and extensible XML execution format for XML-based business process languages. We used a small subset of WS-BPEL to illustrate how an industry standard XML-based programming language can be extended to an XML-based execution format using ideas from process calculi. By also representing the reaction rules as XML we provide an interchangeable format for the semantics and narrowing the gap usually arising between a programming language and its formalisation. The case suggested an interesting extension of BRS to allow for (linear) higher-order reaction rules constrained by tree logics, in this concrete case a subset of XPath, resulting in a kind of *context-dependent* reaction rules. We are currently working on expressing a more general category of higher-order contexts as a Geometry of Interaction [1, 12, 13] construction on the underlying category of bigraphs and show that the general theory of bisimulation congruences for bigraphs can be extended to this setting.

The WS-BPEL process calculus described in the previous sections is just a subset of a WS-BPEL process calculus which has been described and implemented as Distributed Reactive XML in [27]. We have so far only focussed on language primitives found in the XLANG subset. We leave for future work to demonstrate that the flow-graph primitives of the WFDL subset can be represented equally succinct.

The implementation of Distributed Reactive XML so far serves as a proof of concept. However, by representing the business process descriptions, their state and semantics of the process languages as XML and implementing it on top of a distributed peer-to-peer XML storage layer allowing concurrent reactions on shared processes and data, we achieve a middleware supporting many of the features of the ideal scenario described in [7]. We leave for future work to study the relationship between our approach and the approaches surveyed in [7], in particular the Workspaces approach.

References

[1] Samson Abramsky. Retracing some paths in process algebra. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1996.

[2] Thomas Ambus. Multiset discrimination for internal and external data management. Master's thesis, Dept. of Computer Science, University of Copenhagen (DIKU), 2004. URL ⟨`http://www.thomas.ambus.dk/plan-x/msd/`⟩.

[3] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services (version 1.1). Technical report, IBM, Microsoft, SAP, and Siebel Systems, May 2003.

[4] Luca Cardelli. Semistructured computation. In *Proceedings of the 7th International Workshop on Database Programming Languages (DBPL)*, volume 1949 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[5] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[6] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL). Technical report, W3C, January 2006.

[7] Paolo Ciancarini, Robert Tolksdorf, and Franco Zambonelli. Coordination middleware for XML-centric applications. In *Proceedings of 2002 ACM Symposium on Applied Computing (SAC)*, pages 336–343. ACM Press, 2002.

[8] The Workflow Management Coalition. Process definition interface — XML process definition language (version 2.00). Technical Report WFMC-TC-1025, Workflow Management Coalition (WfMC), 2005.

[9] Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. Bilogics: Spatial-nominal logics for bigraphs. 2004.

[10] Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. Bigraphical logics for XML. In *Proceedings of the 13th Italian Symposium on Advanced Database Systems (SEBD)*, pages 392–399, 2005.

[11] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 38–49. ACM Press, 2003.

[12] Jean-Yves Girard. Geometry of interaction I: interpretation of system F. In *Proceedings of Logic Colloquium (1988)*, pages 221–260. North-Holland, 1989.

[13] Jean-Yves Girard. Geometry of interaction II: deadlock free algorithms. In *Proceedings of the International Conference on Computer Logic (COLOG)*, number 417 in Lecture Notes in Computer Science, pages 76–93. Springer-Verlag, 1989.

[14] Mike Havey. *Essential Business Process Modelling*. O'Reilly, 2005.

[15] Fritz Henglein and Henning Niss. Plan-X webpage, 2005. URL ⟨`http://www.plan-x.org/`⟩. (XML Store is part of the Plan-X Project).

[16] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[17] Thomas Hildebrandt and Jacob W. Winther. Bigraphs and (Reactive) XML. Technical Report TR-2005-56, IT University of Copenhagen, 2005.

[18] Thomas Hildebrandt, Henning Niss, Martin Olsen, and Jacob W. Winther. Distributed Reactive XML. In *1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord)*, 2005.

[19] Thomas Hildebrandt, Henning Niss, and Martin Olsen. Formalising business process execution with bigraphs and Reactive XML. In *Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION)*, volume ?? of *Lecture Notes in Computer Science*. Springer, 2006. Accepted for publication.

[20] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge, Computer Laboratory, 2004.

[21] Frank Leymann. Web services flow language (WSFL). Technical report, IBM Software Group, 2001.

[22] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, 2002.

[23] Robin Milner. Bigraphs for Petri nets. In *Lectures on Concurrency and Petri Nets*, number 3098 in Lecture Notes in Computer Science, pages 686–701. Springer-Verlag, 2003.

[24] Robin Milner. Bigraphical reactive systems. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35, 2001.

[25] Robin Milner. Axioms for bigraphical structure. Technical Report UCAM-CL-TR-581, University of Cambridge, Computer Laboratory, 2004.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[27] Martin Olsen. Encoding mobile workflows in Reactive XML. Master's thesis, IT University of Copenhagen, 2006. In Danish.

[28] Kasper Bøgebjerg Pedersen and Jesper Tejlgaard Pedersen. Value-oriented XML Store. Master's thesis, IT University of Copenhagen, 2002. URL ⟨http://www.it-c.dk/people/kasperp/xmlstore/pdf/thesis.pdf⟩.

[29] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[30] Frank Puhlmanm and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In *Proceedings of BPM 2005*, number 2678 in LNCS. Springer-Verlag, 2005.

[31] Christian Stahl. A Petri net semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, jul 2005.

[32] Christian Stefansen. A declarative framework for enterprise information systems. Master's thesis, Dept. of Computer Science, University of Copenhagen (DIKU), 2005. Qualification Report.

[33] Satish Thatte. XLANG: Web services for business process design. Technical report, Microsoft Corporation, 2001.

[34] Robert Tolksdorf. Workspaces: A web-based workflow management system. *IEEE Internet Computing*, september 2002.

[35] Wil M.P van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". *BPTrends*, 3(5):1–11, 2005.

[36] Jacob W. Winther. Reactive XML. Master's thesis, IT University of Copenhagen, 2004.