A Typed Calculus for Querying Distributed XML Documents

Lucia Acciai[1], Michele Boreale[2], and Silvano Dal Zilio[1]

[1] Laboratoire d'Informatique Fondamentale de Marseille
[2] Dipartimento di Sistemi e Informatica, Università di Firenze

There is by now little doubt that XML will succeed as a *lingua-franca* of data interchange on the Web. As a matter of fact, XML is a building block in the development of new models of concurrent applications, often referred to as SOA (for Service-Oriented Architecture) or *Web Services*, where computational resources are made available on a network as a set of loosely-coupled, independent services. This model is characterized by the need to exchange and query distributed, sometimes very large, XML documents. This is the case, for example, when interacting with distributed heterogeneous databases or when processing dynamically generated data, such as those originating from arrays of sensors. These features must be taken into account when designing an effective computational model for SOA.

We study the problems related to querying large, distributed XML documents. We most particularly pay attention to the programming languages problems and, to this end, we define a new formal model for SOA. Our proposal takes the form of a new process calculus in which XML data are processes that can be queried by means of concurrent pattern-matching expressions. In our model, the evaluation of patterns is distributed among locations, in the sense that the evaluation of a pattern at a node triggers concurrent evaluation of sub-patterns at other nodes, and actions can be carried out upon success or failure of patterns. The calculus also provides primitives for gathering intermediate results and orchestrating the evaluation of patterns. Another major feature of our model is the definition of a strong type system based on *regular type expressions*, a model compatible with Document Type Definitions (DTD) and other XML schema languages. The goal of our formal semantics is to provide a sound basis for the implementation of Web Services, to guarantee the soundness of our type system, and to formally prove properties of programs, such as code optimizations.

What we achieve is a functional, strongly-typed programming model for computing over distributed XML documents based on three main ingredients: a semantics defined by an asynchronous process calculus in the style of Milner's π-calculus [13] and concurrent-ML [9]; a model where documents and expressions are both represented as processes, and where evaluation is represented as a parallel composition of the two; a type system based on regular type expressions. Each of these choices is motivated by a feature of the problem: the study of distributed Web Services calls for including concurrency and explicit locations in our model; the need to manipulate large, dynamically generated documents calls for a streamed model of processing; the documents handled by a service should often obey a predefined schema, hence the need to check that queries are well-typed (preferably before they are executed or shipped).

1 Documents, Types and Regular Pattern Expressions

We consider a simplified version of XML such that a document d is an ordered sequence of elements $\langle \mathtt{a}_1 \rangle d_1 \langle /\mathtt{a}_1 \rangle \ldots \langle \mathtt{a}_n \rangle d_n \langle /\mathtt{a}_n \rangle$ (i.e. we leave aside attributes and entities among other

details of the XML specification). In our model, we explicitly take into account that the content of a document may be distributed. We represent each element $\langle \mathtt{a} \rangle d \langle / \mathtt{a} \rangle$ by a term $\langle\, \imath \mapsto \mathtt{a}(\imath_1, \ldots, \imath_n) \,\rangle$ that lists the location $\imath$ where the element is stored, its tag, and the index of the locations where the elements in d can be found. For instance, the document $\langle \mathtt{a} \rangle \langle \mathtt{b} / \rangle \langle \mathtt{c} / \rangle \langle / \mathtt{a} \rangle$ is represented by the parallel composition $(\nu \imath_1, \imath_2)\big(\langle\, \imath \mapsto \mathtt{a}(\imath_1, \imath_2) \,\rangle \mid \langle\, \imath_1 \mapsto \mathtt{b}(\,) \,\rangle \mid \langle\, \imath_2 \mapsto \mathtt{c}(\,) \,\rangle\big)$, where the notation $(\nu \imath) P$ means that the location $\imath$ is known only to P.

In our calculus, a type is a regular (tree) expression that constraints the occurrences of elements in a valid document. Types take the form of a tree grammar. For instance, the type $A = \mathtt{a}[A] + \mathit{Empty}$ matches either the empty document or documents $\langle \mathtt{a} \rangle d \langle / \mathtt{a} \rangle$ where d is of type A. The type $B = (\mathtt{a}[\mathit{All}] + \mathtt{b}[\mathit{All}])*$ matches sequences of elements tagged by $\mathtt{a}$ or $\mathtt{b}$ (the type All matches every XML document). Following existing functional languages for XML, see e.g. XDuce [11] or the review in [5], we define patterns by extending types with capture variables. Patterns are used for inspecting documents and extracting information. For example, the patterns p_1, p_2 defined below can be used to extract the content of every element tagged $\mathtt{b}$ in a document of type B, and store these values into (the reference) x.

$$p_1(x) = (\mathtt{a}[p_1(x)] + \mathtt{b}[p_2(x)])* \quad \text{and} \quad p_2(x) = (\mathtt{a}[p_1(x)] + \mathtt{b}[p_2(x)]) * \mathit{as}\ x$$

We can use patterns in expressions of the form $\mathit{let}\ y = \big(\mathit{try}\ d\ p(\boldsymbol{x})\big)\ \mathit{in}\ e$, meaning that the filtering of d by p is executed concurrently with the evaluation of e. In this example, y is bound to the identity/location of the thread that executes the pattern-matching (this value can be tested to check whether the pattern-matching has failed or is finished).

2 The Language

The core of our calculus is a first-order functional language equipped with constructs for building and filtering documents. The basic components of the calculus are *locations* $\imath, \ell$, which identify nodes in the network and contain resources; *resources*, which are document nodes or active filters; *expressions*, which include calls to functions or patterns, operation on references, ...; and *processes* built from parallel composition of resources and let-expressions.

results:	u, v	$::= x \mid \ell \mid (\imath_1, \ldots, \imath_n)$	(locations and indexes)
expressions:	e	$::= u \mid \mathtt{a}[u] \mid u \cdot v$	(document expressions)
		$\mid \mathit{newref}\ u \mid\ !u \mid u := v$	(references)
		$\mid \mathit{try}\ u\ p(\boldsymbol{v}) \mid \mathit{wait}\ u?(x)\ e_1\ e_2$	(pattern-matching)
		$\mid f(\boldsymbol{u}) \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2$	(functions)
processes:	P, Q	$::= e \mid \mathit{let}\ x = P\ \mathit{in}\ Q \mid \langle\, \imath \mapsto \mathit{ref}\ u \,\rangle \mid \langle\, \imath \mapsto \mathtt{a}(u) \,\rangle \mid P \mid Q \mid (\nu \imath) P$	

We do not have enough space to define the operational semantics of our calculus, but we try to convey some intuitions by studying a simple example of process.

$$P \ =_{\text{def}}\ \langle\, \ell \mapsto \mathit{ref}\ u \,\rangle \mid \langle\, \imath \mapsto \mathtt{b}(\,) \,\rangle \mid \mathit{let}\ x = \mathit{try}\ d\ p(\ell, \ell)\ \mathit{in}\ \big(\mathit{wait}\ x?(z)\ e_1\ e_2\big) \mid \big(\ell := \mathtt{a}[\imath]\big)$$

Assume that the pattern $p(x, y)$ collects in x (resp. y) the elements tagged with $\mathtt{a}$ (resp. $\mathtt{b}$). The computation of P is as follows. The *try* expression results in all elements tagged $\mathtt{a}$ or $\mathtt{b}$ in d to be referenced by ℓ. Concurrently, two new processes are created that (1) waits for the *try*-expression to succeed or fail, with continuation $e_1\{z \leftarrow d\}$ or $e_2\{z \leftarrow d\}$; and (2) appends to ℓ a location storing an element of the form $\langle \mathtt{a} \rangle \langle \mathtt{b} / \rangle \langle / \mathtt{a} \rangle$.

An important feature of our calculus is that every pattern is strongly typed: its type is the regular expression obtained by erasing capture variables. Likewise we can type locations, expressions and processes using a combination of regular type expressions with functional and *ref* types. (The definition of the type system escapes the scope of this short abstract.) As it is often the case with typed calculi, the first important property we need to prove is that well-typedness of processes is preserved by reduction: in a long version of this paper, we formally define the static and dynamic semantics of the calculus and prove a *subject reduction* theorem. The proof of this property is quite involved since it is not possible to reason on a whole document at once: its content is scattered across distinct resource locations. This complexity reflects actual problems imposed by our problem since distributed documents can never be checked locally.

3 Conclusions and related work

We define a formal model for computing over large, distributed XML documents based on a new typed process calculus. Recent works have addressed the problem of integrating XML and π-calculus [1, 3, 10]. In all these proposals, documents are first class values exchanged in messages, which make these calculi inappropriate in the case of very large or dynamically generated data. At the opposite, we consider documents as special kind of processes that can be randomly accessed through the use of indexes.

Our work may also be compared with proposals for filtering and querying streams of XML elements. We can identify two main approaches in this case: the first is to provide an efficient single-pass evaluator, working with one query at a time [4, 7, 12]; the second is in the context of peer-to-peer and event-notification systems, where XML streams need to be filtered by a large number of queries [2, 6, 8]. Our approach follows the first direction: a pattern is equivalent to a query and a *try*-statement applies one pattern at a time. But we can point out some differences that arise in our work. Instead of basing our system on XPath or XQuery, we extend the approach introduced by functional languages such as XDuce and define distributed regular pattern expressions. As a byproduct, we also provide a possible semantics for a concurrent extensions of languages based on XDuce.

References

1. L. Acciai and M. Boreale. XPi: a typed process calculus for XML messaging. In *Proc. of FMOODS*, 2005.
2. M. Altinel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination information. In *Proc. of VLDB*, 2000.
3. G. Bierman and P. Sewell. Iota: A concurrent XML scripting language with applications to Home Area Networking. TR 577, Computer Laboratory, Cambridge, 2003.
4. F. Bry, T. Furche, and D. Olteanu. An efficient single-pass query evaluator for XML data structure. TR PMS-FB-2004-1, Computer Science Institute, Munich, 2004.
5. G. Castagna. Pattern and types for querying XML documents. In *Proc. of DBPL*, XSYM, 2005.
6. C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. of ICDE*, 2002.
7. S.S. Chawathe and F. Peng. XPath Queries on Streaming Data. In *Proc. of SIGMOD*, 2003.
8. Y. Diao, P. Fisher, and M.J. Franklin. Yfilter: efficient and scalable filtering of XML documents. In *Proc. of ICDE*, 2002.
9. W. Ferreira, M. Hennessy and A.S. Jeffrey A theory of weak bisimulation for core CML J. Functional Programming 8(5), pp. 447-491, 1998.
10. P. Gardner and S. Maffeis. Modeling dynamic Web data. In *Proc. of DBPL*, 2003.
11. H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL*, 2001.
12. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A tranducer-based XML query processor In *Proc. of VLDB*, 2002.
13. R. Milner. Communicating and Mobile Systems: The π-calculus. CUP. 1999.