Model Checking XML Manipulating Software

Xiang Fu
fuxiang@cs.ucsb.edu

Tevfik Bultan
bultan@cs.ucsb.edu

Jianwen Su
su@cs.ucsb.edu

Department of Computer Science
University of California
Santa Barbara, CA 93106-5110

ABSTRACT

The use of XML as the de facto data exchange standard has allowed integration of heterogeneous web based software systems regardless of implementation platforms and programming languages. On the other hand, the rich tree-structured data representation, and the expressive XML query languages (such as XPath) make formal specification and verification of software systems that manipulate XML data a challenge. In this paper, we present our initial efforts in automated verification of XML data manipulation operations using the SPIN model checker. We present algorithms for translating (bounded) XML data and XPath expressions to Promela, the input language of SPIN. The techniques presented in this paper constitute the basis of our Web Service Analysis Tool (WSAT) which verifies LTL properties of composite web services.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

General Terms

Verification, Design

Keywords

Model Checking, SPIN, Promela, Web Service, XML, XML Schema, MSL, XPath.

1. INTRODUCTION

Web based software systems (e.g. web services) are becoming increasingly important partly due to the wide use of the Web in electronic commerce. Errors in such systems, where multi-million dollar transactions are carried out, can be very costly; ad-hoc repairs after failure are not acceptable. Static analysis techniques and especially model checking can be very valuable in ensuring the correctness and robustness of such systems before they are deployed.

It is generally agreed that messages exchanged among web based systems should be in the XML [21] format. For example, almost all web service standards (e.g. WSDL [18], BPEL4WS [1], WSCI [20], OWL-S [17]) are built on XML and related standards including XML Schema [23] and XPath [22]. The rich tree-structured data representation of XML and powerful XPath expressions, however, impede direct application of model checking techniques to the verification of Web based systems. Earlier efforts to verify web services (e.g. [6, 15, 12]) basically focus on only the control flows by abstracting away the XML data semantics during analysis.

This paper presents our initial efforts in formal specification and verification of software systems with XPath based manipulation of (bounded) XML data. The techniques presented in this paper constitute the basis of our Web Service Analysis Tool (WSAT) [10, 19] which can verify Linear Temporal Logic (LTL) properties of conversation protocols [7] and interacting BPEL4WS [1] web services [8]. Clearly, these techniques can also be used for verification of other types of software systems that exchange XML data.

We use SPIN [11] as a back-end model checker in verification of XML data manipulation operations. We developed algorithms for translating XML data types and XPath expressions to Promela, the input language of SPIN. Our handling of XML data manipulation consists of two parts: (1) a mapping from XML Schema to the type system of Promela, and (2) a translation algorithm which generates Promela code for an XPath expression. The type mapping is straightforward; however, the translation of XPath expressions is not trivial. We implemented the translation algorithms presented in this paper as a part of WSAT.

Our use of SPIN as the back-end model checker is based on the following considerations: (1) Promela supports arrays which is very useful in translating XML Schema data types. (2) The communication channels in Promela enables us to model the asynchronous communication among web services [8]. However, SPIN is an explicit-state model checker, and may not scale to large data domains due to state-space explosion. In the future we plan to investigate the use of symbolic model checking techniques in verification of XML data manipulation.

In [15] verification and composition of web services are investigated using a Petri Net model. In [6], web service compositions are specified using message sequence charts, modeled using finite state machines and analyzed using the LTSA model checker. These earlier efforts on verification of Web based software systems mostly concentrate on analysis of the control flows. Our techniques for handling XML data, however, enable verification of properties relating to data manipulation. This enables us to analyze Web based software systems at a greater level of detail without ad-hoc data abstractions. The idea of employing back-end model checkers for verification of an expressive language is used in other verification tools such as Bandera [4].

JWIG project extends the Java language with high-level features

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

for web service programming such as dynamic construction of XML documents [5]. To ensure that the generated XML document is consistent with the message format (declared using DSD [14], a type system similar to XML Schema), JWIG provides static analysis for a set of pre-defined properties. The verification problem considered in this paper (and in WSAT) is rather different: we consider the relationships (expressed in temporal logic) between *multiple* XML messages during the execution of a web service. Also, we focus on XPath expressions which are not part of JWIG.

The techniques presented in this paper apply to bounded XML data only, where the number of children of an XML node is always bounded. Unbounded XML Schema types, and various fragments of XPath can be captured using unranked tree automata [13, 16]. While the unranked tree automata model overcomes the problem of boundedness, the data semantics of leaf value nodes are lost in the modeling. For example, the fragment of XPath studied in [13] does not allow arithmetic constraints in qualifiers, and it only reasons about the structure of an XML document.

This paper is organized as follows. Section 2 reviews XML related standards that are essential for systems that exchange XML data. Formal models are established for these standards to facilitate the technical discussion later. Section 3 introduces the mapping from MSL (a theoretical model for XML Schema) to Promela, and the translation algorithm from XPath to Promela is presented in Section 4. Section 5 discusses the application of the presented techniques to the verification of web services. Section 6 concludes the paper.

2. XML RELATED STANDARDS

In this section we present the syntax and semantics of XML, MSL, and XPath. The translation algorithms presented in the following sections build on the definitions given in this section.

2.1 XML

Extensible Markup Language (XML) is a markup language used for describing data [21]. As the de facto universal data transfer format over the Internet, XML plays a central role in specifying semi-structured data in a way that is platform and language neutral, and to a degree, self-explanatory. XML Schema [23] provides the type system for XML, i.e., it is used to specify the expected data organization of an XML document. XPath [22], one of the most popular XML query languages, is used to navigate through an XML document and to access its components.

Similar to HTML, all XML documents are structured using tags, which are written as `<tag>` followed by `</tag>`. However, tags in XML describe the content of the data rather than the appearance. Fig. 1(a) shows an XML document containing the data for a `Register` message sent from an investor to register for a stock analysis service (a description of the service is provided in Section 5). The XML document consists of a string containing the identification of the investor, a list of stock identifiers that the investor is interested in, and payment information.

XML documents can be modeled as trees where each internal node corresponds to a tag and leaf nodes correspond to basic type values. The document in Fig. 1(a) corresponds to the tree in Fig. 1(b).

In the following we introduce a formal representation for XML documents. One simplification we make here is to omit the tag attributes used in XML. Since a tag attribute can be regarded as a leaf node that is a child of the corresponding tag node, this simplification does not impair the expressive power of our model.

DEFINITION 2.1. An *XML document* is a quadruple $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$ where

1. l is a list of *labels* where each label can either be an internal node tag, or a leaf node value with a basic type (such as boolean, integer or string). We denote i-th node of l with $l[i]$ (indices start from 1).
2. n is the size of l.
3. $\mathrm{p} : [1, n] \rightarrow [0, n-1]$ is a *parent* function such that
   (a) $\mathrm{p}(1) = 0$, and
   (b) for each $1 < i \leq n$, $1 \leq \mathrm{p}(i) < i$

   We define p^* as the transitive and reflexive closure of p.
4. $\mathrm{r} : [1, n] \rightarrow [1, n]$ is a *range* function where
   (a) $\mathrm{r}(i) \geq i$ for each $i \in [1, n]$, and
   (b) for each $i \leq j \leq \mathrm{r}(i)$, $i \in \mathrm{p}^*(j)$, and for each $j \notin [i, \mathrm{r}(i)]$, $i \notin \mathrm{p}^*(j)$.

Given a node at index i, $\mathrm{p}(i)$ points to its parent node. Since the root has no parent, we define $\mathrm{p}(1) = 0$. For each node i, $\mathrm{r}(i)$ denotes the maximum index of the nodes in the subtree of node i. Note that constraints on p and r guarantee that l is the pre-order traversal of the document tree of $\mathcal{X}$.

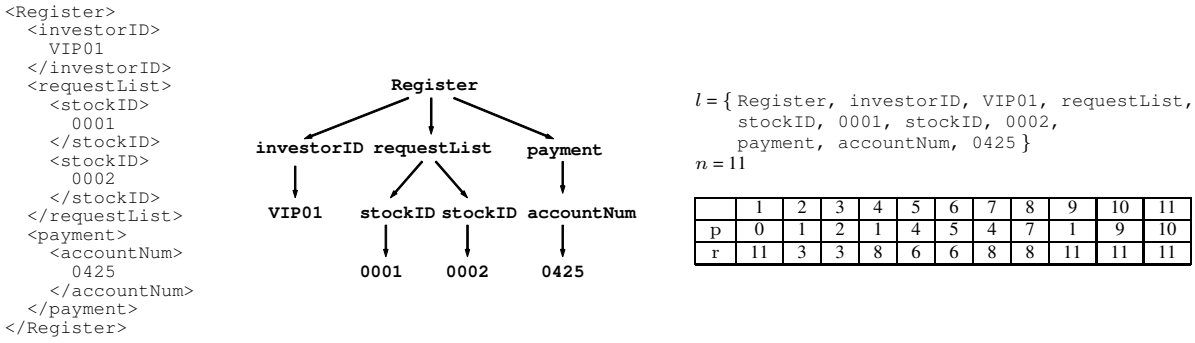
EXAMPLE 2.1. Fig. 1(c) is the quadruple representation of the XML document in Fig. 1(a). Obviously the list l is the pre-order traversal of the tree in Fig. 1(b), and parent function p and range function r describe the tree structure. For example, the subtree starting from node `requestList` spans over five nodes; hence the range function $\mathrm{r}(4) = 8$. ∎

Definition 2.1 can be extended to describe a *tree sequence*, when restriction "$1 \leq \mathrm{p}(i)$" in item 3(b) is modified to "$0 \leq \mathrm{p}(i)$". In a tree sequence, we call each node whose parent node is 0 a *root node*. We introduce a *split* operator that splits a tree sequence into two tree sequences, and an *extract* operator that generates a tree sequence from a single XML document tree by extracting its contents.

DEFINITION 2.2. Given an XML tree sequence $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$, a *split* at integer s can be applied to $\mathcal{X}$ if node s is a root node and $s \neq 1$. The result is two tree sequences $\mathcal{X}_1 = (l_1, n_1, \mathrm{p}_1, \mathrm{r}_1)$ and $\mathcal{X}_2 = (l_2, n_2, \mathrm{p}_2, \mathrm{r}_2)$ where

1. $l_1 = l[1, s-1]$ and $l_2 = l[s, n]$.
2. $n_1 = s-1$, and $n_2 = n-s+1$.
3. p_1 coincides with p on the domain $[1, s-1]$, and
$$\mathrm{p}_2(i) = \begin{cases} \mathrm{p}(i+s-1)-s+1 & \text{if } \mathrm{p}(i+s-1) \neq 0 \\ 0 & \text{if } \mathrm{p}(i+s-1) = 0 \end{cases}$$
   for each $i \in [1, n-s+1]$.
4. r_1 coincides with r, but is restricted to the domain $[1, s-1]$, and $\mathrm{r}_2(i) = \mathrm{r}(i+s-1)-s+1$ for each $i \in [1, n-s+1]$.

Given an XML tree sequence which has at least m root nodes, for any $k \leq m$ we can split the tree sequence into k sequences, by consecutively applying the split operator $k-1$ times to the second part of the result of the previous split. We call this operation a k-split.



	1	2	3	4	5	6	7	8	9	10	11
p	0	1	2	1	4	5	4	7	1	9	10
r	11	3	3	8	6	6	8	8	11	11	11

(a) (b) (c)

Figure 1: An XML document (a), the corresponding tree (b), and its formal representation (c)

DEFINITION 2.3. Given a single XML tree $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$, the *extract* operator generates a tree sequence $\mathrm{extract}(\mathcal{X}) = (l', n', \mathrm{p}', \mathrm{r}')$ where $l' = l[2, n]$, $n' = n-1$, and for each $i \in [1, n']$, $\mathrm{r}'(i) = \mathrm{r}(i+1)-1$ and $\mathrm{p}'(i) = \mathrm{p}(i+1)-1$.

EXAMPLE 2.2. If we apply the exact operator to the XML tree in part (c) of Fig. 1, we get the XML tree sequence $\mathcal{X}' = (l', n', \mathrm{p}', \mathrm{r}')$ where

l' = { investorID, VIP01, requestList, stockID, 0001, stockID, 0002, payment, accountNum, 0425 }
n' = 10

	1	2	3	4	5	6	7	8	9	10
p′	0	1	0	3	4	3	6	0	8	9
r′	2	2	7	5	5	7	7	10	10	10

Note that, the tree sequence $\mathcal{X}'$ can be split at 3 and 8, and there exists a 3-split for $\mathcal{X}'$. ∎

2.2 XML Schema and MSL

XML provides a standard way to exchange data over the Internet. However, the parties that exchange XML documents still have to agree on the *type* of the data, i.e., what are the tags that will appear in the document, in what order, etc. XML Schema [23] is a language for defining XML data types. In this paper, we focus on a subset of XML Schema, for example, we do not handle unordered sequence types. Model Schema Language (MSL) [2] is a compact formal model that captures most features of XML Schema. We use a simplified version of MSL with type expressions defined as follows:

$$g \rightarrow b \mid t\,[\,g_0\,] \mid g_1\{m,n\} \mid g_1\,,\ldots,g_k \mid g_1 \mid \ldots \mid g_k$$

Here $g, g_0, g_1, \ldots, g_k$ represent MSL types, b is a basic data type such as string, integer or boolean, t is a tag, and m and n are two positive integers where $m \leq n$. Intuitively, the semantics of the above MSL type expressions can be summarized as follows: $t\,[\,g_0\,]$ denotes a type with a root node labeled with t and children with types that match the sequence of MSL types represented by g_0; $g_1\{m,n\}$ denotes a sequence of size at least m and at most n where each member is of type g_1; $g_1\,,\ldots,g_k$ denotes an ordered sequence where the first member is of type g_1, the second member is of type g_2, and so on; and, $g_1 \mid \ldots \mid g_k$ denotes a choice among types g_1 to g_k. To simplify our presentation, we will assume that the types $g_1, \ldots, g_k$ that appear in $g_1\,,\ldots,g_k$ and $g_1 \mid \ldots \mid g_k$ are derived by the rules "$g \rightarrow b$" or "$g \rightarrow t\,[\,g_0\,]$".

Similar to XML, we can define a "parent function" for MSL types. Given two MSL types g and g_i, $\mathrm{p}(g_i) = g$ if there exists a g' such that either of the following two conditions are satisfied: 1) $g \rightarrow t[g'] \wedge g' \rightarrow g_1\,,\ldots,g_i\,,\ldots,g_k$, or 2) $g \rightarrow t[g'] \wedge g' \rightarrow g_1 \mid \ldots \mid g_i \mid \ldots \mid g_k$. We associate an attribute called "*tag*" with each MSL type g. If g is derived from the rule $g \rightarrow t[g_0]$, then $g.tag = t$; otherwise *tag* is `null`.

Formally, an XML document tree sequence $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$ is an *instance* of an MSL type g if one of the following holds:

1. when $g \rightarrow b$: $n = 1$ and $l[1]$ is a leaf node value and its type is b.
2. when $g \rightarrow t\,[\,g_0\,]$: $\mathcal{X}$ is a single XML document tree where $l[1] = t$ and $\mathrm{extract}(\mathcal{X})$ is an instance of g_0.
3. when $g \rightarrow g_1\{m,n\}$: there exists a k-split on $\mathcal{X}$ for some integer $m \leq k \leq n$ such that the resulting tree sequences $\mathcal{X}_1$, ..., $\mathcal{X}_k$ are all instances of g_1.
4. when $g \rightarrow g_1\,,\ldots,g_k$: there exists a k-split of $\mathcal{X}$, such that the resulting tree sequences $\mathcal{X}_1, \ldots, \mathcal{X}_k$ are instances of $g_1, \ldots, g_k$ respectively.
5. when $g \rightarrow g_1 \mid \ldots \mid g_k$: $\mathcal{X}$ is an instance of g_i for some integer $i \in [1, k]$.

EXAMPLE 2.3. It is easy to verify that the XML document `Register` in Fig. 1 is an instance of the following MSL type.

```
Register[
 investorID[string],
 requestList[ stockID[int]{1,3} ],
 payment[ creditCard[int] | accountNum[int] ]
]
```

∎

2.3 XPath

In order to write specifications or programs that manipulate XML documents we need an expression language to access values and nodes in XML documents. We use a subset of XPath [22] to navigate through XML trees and return the answer nodes. The fragment of XPath we use consists of the following operators: the child axis (`/`), the descendant axis (`//`), self-reference (`.`), parent-reference (`..`), basic type test (`b()`), node name test (t), wildcard (`*`), and predicates (`[]`). An *XPath expression* is defined with the following grammar

$$\begin{aligned} exp &\rightarrow p \mid \texttt{const} \mid \texttt{op}\ exp \mid exp\ \texttt{op}\ exp \\ p &\rightarrow r \mid /r \mid //r \\ r &\rightarrow s \mid r/s \mid r//s \\ s &\rightarrow . \mid .. \mid n^{?}(\texttt{[}exp\texttt{]})^{*} \mid \texttt{position()} \mid \texttt{last()} \\ n &\rightarrow b\texttt{()} \mid t \mid * \end{aligned}$$

where $n^{?}$ denotes n or empty string and $(\texttt{[}exp\texttt{]})^{*}$ denotes zero or more repetition of `[`exp`]`.

In the above syntax rules, an expression on basic types (such as boolean, integer, and string) can be constructed by combining

XPath location paths (represented by p) and constant values (represented by `const`) with operators on basic types (represented by `op`). There are two types of location paths: relative location paths and absolute location paths. An absolute location path starts with `/` or `//`. A relative location path (represented by r) consists of a list of steps (represented by s) which are connected with `/` or `//`. The steps in a relative location path are evaluated from left to right. A step can be a self-reference (`.`), a parent-reference (`..`), or a more complex form which consists of a node test (n) and a sequence of predicates of the form `[`exp`]`. A node test n has three possible forms: type test (b`()`), name test (t), and wildcard match ($*$). Finally, a step can be a function call such as `position()` or `last()` with the following restriction: Function calls can only appear as the last step of a location path.

Formally, an XPath expression accepts inputs of the form (c, d) where *context* c is a set of node indices in some XML document $\mathcal{X}$, and d is either a single node in $\mathcal{X}$ or a set of values with the same basic type. The set of node indices in the input context must be in ascending order with no repetition. The output of an XPath expression is a set of nodes in the same XML document used in the input, or a set of values. For $\mathcal{X}(l, n, \mathrm{p}, \mathrm{r})$, let N be the domain of all node indices in $\mathcal{X}$ (i.e., $[1, n]$), and DOM be the domain of all leaf node values. Then the semantics of an XPath expression exp (as well as a step s, a node test n, and a location path r) can be defined as a function:

$$exp : 2^N \times (N \cup 2^{DOM}) \rightarrow 2^N \cup 2^{DOM}.$$

Before we formally define the semantics of XPath expressions, we will give some example expressions and the results of evaluating them below. To distinguish node indices from other values we write node indices in bold characters.

EXAMPLE 2.4. Consider the input $(\{\mathbf{1}\}, \mathbf{1})$, where $\mathbf{1}$ is the root node of the XML document presented in Fig. 1, and the following XPath expressions

- `investorID`,
- `//stockID[position() = 2]/int()`, and
- `//stockID/int() = 0002`

The results are $\{\mathbf{2}\}$, $\{0002\}$, and {`false, true`}, resp. ∎

We now present the formal semantics of XPath expressions. We will give the semantics of a node test first, and then a step, and then a location path, and finally an entire XPath expression.

Consider a node test n and an input (c, d) where c is a set of node indices and d a node in the XML document $\mathcal{X}(l, n, \mathrm{p}, \mathrm{r})$, the results of $n(c, d)$ is defined as follows:

1. when $n \rightarrow$ `b()`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d \ \wedge \ l[d']$ is of type `b`$\}$.
2. when $n \rightarrow$ `t`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d \ \wedge \ l[d'] = t\}$.
3. when $n \rightarrow$ `*`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d\}$

Basically, rules 1 and 2 select the children nodes of the input node d by their types and tags (resp.), and rule 3 simply returns all the children nodes. Finally, when input d is a set of values, $n(c, d)$ returns the empty set $\emptyset$.

The definition of a step s is similar. Given a step s and input (c, d), when d is a set of values, s simply returns $\emptyset$, and when d is a node of $\mathcal{X}$ the result is defined as follows:

1. when $s \rightarrow$ `.` : $s(c, d) = \{d\}$.
2. when $s \rightarrow$ `..` : $s(c, d) = \{\mathrm{p}(d)\}$.
3. when $s \rightarrow [exp]$:

$$s(c,d) = \begin{cases} \{d\} & \texttt{true} \in exp(c,d) \text{ and} \\ & \quad exp \text{ is a boolean expression} \\ \{d\} & exp(c,d) \neq \phi \text{ and} \\ & \quad exp \text{ is a location path} \\ \emptyset & \text{otherwise.} \end{cases}$$

4. when $s \rightarrow n\,[exp_1] \ldots [exp_k]$: $s(c, d) = p'(c, d)$ where

$$p' = n \ / \ [exp_1] \ / \ [exp_2] \ / \ \ldots \ / \ [exp_k]$$

5. when $s \rightarrow$ `last()`: $s(c, d) = \{|c|\}$
6. when $s \rightarrow$ `position()`: $s(c, d) = \{\text{position of d in c}\}$

In rules 1 and 2, the handling of self reference and parent reference is straightforward. For the third rule, the step either returns the singleton set $\{d\}$ or an empty set, depending on the evaluation of the predicate. Note that when a location path is used as a predicate, it evaluates to `true` if it returns a non empty set of nodes. Finally, the evaluation of a step which consists of node test plus a series of predicates is reduced to that of an equivalent location path. For example, the step `stockID[position()=2]` is equivalent to a location path `stockID/[position()=2]`. Given input (c, d), function call `last()` simply returns the singleton set which contains the size of context c; `position()` returns the position of d in c where position is counted starting from 1.

Given a relative path $r \rightarrow r_1 / s$, and an input (c, d),

$$r(c, d) = \bigcup_{d' \in r_1(c,d)} s(r_1(c, d), d'). \tag{1}$$

According to the above formula the steps of a relative location path are executed one by one from left to right. Note that the context of each step is the result of the previous step. For the case where $r \rightarrow r_1 // s$, we replace $r_1(c, d)$ in Equation 1 with the following set $\{n \mid \mathrm{p}^*(n) = n' \text{ for some } n' \in r_1(c, d)\}$. For example, given the XML document in Fig. 1 and the input $(\{\mathbf{4}\}, \mathbf{4})$, the first step and the descendants operator of the location path `stockID//[position()=3]/int()` produces the new context $\{\mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}\}$, and the later steps generates the result $\{0001\}$.

An absolute location path is defined based on a relative location path. For an absolute location path $p \rightarrow /r$, and input (c, d), $p(c, d) = r(\{\mathbf{1}\}, \mathbf{1})$ where $\mathbf{1}$ is the root. When $p \rightarrow //r$, $p(c, d) = p'(\{\mathbf{1}\}, \mathbf{1})$ where $p' =$ `.//`r.

Finally, the semantics of an XPath expression on basic types in the form $exp \rightarrow exp_1$ `op` exp_2 is defined as follows:

$$exp(c,d) = \{v \mid v = v_1 \texttt{ op } v_2 \ \wedge \ v_1 \in exp_1(c,d) \ \wedge \ v_2 \in exp_2(c,d)\}.$$

Basically, it computes the results of all possible combinations from the value sets of the two operands exp_1 and exp_2. Unary operators are handled similarly. We assume that operation `op` is either a relational operator ($=, \neq, <, >\leq, \geq$), or an arithmetic operator ($+, -, *, /, \%$), or a boolean operator ($\wedge, \vee, \neg$). Note that, when used as a condition, a boolean XPath expression evaluates to `true` if its result set contains at least one `true` value.[1]

[1]XPath 2.0 (working draft) has a more delicate handling for this scenario. There are two sets of arithmetic/comparison operators: one to support the XPath 1.0 semantics (presented in this paper); the other will raise a type error when any operand contains more than one value. It is not hard to support the second semantics with our approach.

```
typedef t1_investorID{
 mtype stringvalue;
}

typedef t2_stockID{
 int intvalue;
}
typedef t3_requestList{
 t2_stockID stockID [3];
 int stockID_occ;
}
typedef t4_accountNum{
 int intvalue;
}
typedef t5_creditCard{
 int intvalue;
}
mtype {m_accountNum,
       m_creditCard}
typedef t6_payment{
 t4_accountNum accountNum;
 t5_creditCard creditCard;
 mtype choice;
}
typedef Register{
 t1_investorID investorID;
 t3_requestList requestList;
 t6_payment payment;
}
```

Figure 2: Promela translation of Example 2.3

3. FROM MSL TO PROMELA

In this section we focus on mapping types in MSL to Promela. We present an example translation in Fig. 2, and the translation algorithm is given in Fig. 3.

Fig. 2 is the Promela translation of the MSL type given in Example 2.3. Clearly each MSL basic type has a straightforward mapping to Promela. For example, `int` and `boolean` are mapped to Promela type `int` and `bool` respectively. MSL type `string` is mapped to `mtype` (enumerated type) in Promela, e.g., the leaf node value of `investorID`. In the XPath translation, which will be explained later, all string constants will be collected, declared, and used as symbolic constants of `mtype`. We assume strings are used solely as constants, and we do not expect any operators to change values of these `mtype` variables.

Translation of complex MSL types is more complicated. Generally, each MSL complex type is translated into a corresponding `typedef` (record) type in Promela. For example, the `Register` type in Example 2.3 is mapped into Promela declaration `typedef Register`, and the intermediate type `requestList` inside `Register` is translated into `typedef t3_requestList`. Prefixes such as `t3_` are added to prevent name collisions for intermediate types. Since each intermediate MSL type is a child of its parent type, in the Promela type declaration for its parent type, it has a corresponding attribute definition. For example, the statement "`t3_requestList requestList`" defines the attribute `requestList` in `typedef Register`. When an intermediate MSL type has multiple occurrences, e.g., the `stockID` element, it is defined as an array with its max occurrence as the array size. In addition, an additional variable (e.g. `stockID_occ`) is defined in its parent type to record its actual occurrence. For the MSL types constructed using the choice operator `|`, a variable `choice` is used to record the actual type chosen in an XML instance of the MSL type (e.g., the `choice` attribute declared in `t6_payment`).

In Fig. 3 we present a procedure `tr`, which takes an MSL type declaration g as its input, and generates two strings as its output. The first string, i.e., `ret[1]`, contains the type declaration for g (as well as all the necessary type declarations for its intermediate types). The second output is the attribute definition for g, if g is an intermediate type. For example, when the procedure is called for intermediate type `requestList`, `ret[1]` contains the declaration of `t2_stockID` and `t3_requestList`, and `ret[2]` contains "`t3_requestList requestList;`". (We do not show the generation of separator "`;`" in Fig. 3, however, it can be handled easily). As shown in Fig. 3, the function body of `tr` processes the input MSL type declaration recursively according to the syntax rules. Note that, it properly handles the issues such as array declaration for types with multiple occurrences and complex types constructed using choice operator.

```
// ret[1]: type declaration for g, including intermediate types.
// ret[2]: attribute definition for the input g if g is intermediate.
function tr(g: MSL): String[2]
begin
  String ret1, ret2;
  switch case
    g → b :
      ret1 = null;
      ret2 = b + " " + b + "value";
    g → t[g0] :
      if (g is an intermediate type) then
        type = generate a unique name;
        ret1 = tr(g0)[1] + "typedef " + type + "{" + tr(g0)[2] + "}";
        ret2 = type + " " + t;
      else
        ret1 = tr(g0)[1] + "typedef " + t + "{" + tr(g0)[2] + "}";
        ret2 = null;
      end if
    g → g1{m, n} :
      ret1 = tr(g1)[1];
      ret2 = tr(g1)[2] + "["+n+"]" + "int " + g1.tag +"_occ"
    g → g1, g2, . . . , gk :
      ret1 = tr(g1)[1] + tr(g2)[1] + . . . + tr(gk)[1];
      ret2 = tr(g1)[2] + tr(g2)[2] + . . . + tr(gk)[2];
    g → g1 | g2 | . . . | gk :
      ret1 = tr(g1)[1] + tr(g2)[1] + . . . + tr(gk)[1] +
             "mtype {"+ "m_"+g1.tag + . . . + "m_"+gk.tag + "}";
      ret2 = tr(g1)[2] + tr(g2)[2] + . . . + tr(gk)[2] + "mtype choice";
  end switch
  return (ret1, ret2)
end
```

Figure 3: Translation from MSL to Promela

4. FROM XPATH TO PROMELA

In this section we present the translation algorithm from XPath to Promela. We start with a brief discussion of the use of XPath expressions in XML manipulating software, then we study a motivating example, and finally we present the translation algorithm.

Consider the use of XPath in languages with XML data manipulation such as BPEL4WS and WSCI. There are basically two types of usage: 1) boolean XPath expressions are used in branch or loop conditions, and 2) location paths and arithmetic expressions are used on the left and right hand sides of assignment statements, respectively. We handle these two cases separately since the semantics of XPath expressions can depend on the context they are used. For example, when a location path is used as a boolean condition its meaning is different than the case where it is used on the left hand side of an assignment. Since the implementation of these two cases are similar, in the remainder of this paper, we concentrate only on the translation of boolean XPath expressions.

4.1 A Motivating Example

Consider the following XPath boolean expression where XML variable `register` is of MSL type `Register` as defined in Example 2.3, and the MSL type of variable `request` consists of a single child `stockID` (in XPath the prefix `$` is used to denote variable names):

```
$request//stockID/int()=
$register//stockID[int()>5][position()=last()]/int()   (2)
```

An XPath expression following a variable name is evaluated on the value of the variable (which is an XML document) starting with the context $(\{\mathbf{1}\}, \mathbf{1})$, where $\mathbf{1}$ is the root node of the corresponding XML document. The above expression queries whether in the XML document `register` the last `stockID` which has a value greater than 5 is equal to the `stockID` of `request`. Its corresponding Promela translation is shown in Fig. 4.

```
1  /* result of the XPath expression */
2  bool bResult = false;
3  /* results of the predicates 1, 2, and 1 resp. */
4  bool bRes1, bRes2, bRes3;
5  /* index, position(), last(), index, position() */
6  int i1, i2, i3, i4, i5;
7
8  i2=1;
9  /* pre-calculate the value of last(), store in i3 */
10 i4=0; i5=1; i3=0;
11 do
12 :: i4 < v_register.requestList.stockID_occ
13   ->
14   /* compute first predicate */
15   bRes3 = false;
16   if
17   :: v_register.requestList.stockID[i4].intvalue>5
18      -> bRes3 = true
19   :: else -> skip
20   fi;
21   if
22   :: bRes3 -> i5++; i3++;
23   :: else -> skip
24   fi;
25   i4++;
26
27 :: else -> break;
28 od;
29 /* translation of the whole expression */
30 i1=0;
31 do
32 :: i1 < v_register.requestList.stockID_occ
33    ->
34    /* first predicate */
35    bRes1 = false;
36    if
37    :: v_register.requestList.stockID[i1].intvalue>5
38       -> bRes1 = true
39    :: else -> skip
40    fi;
41    if
42    :: bRes1 ->
43       /* second predicate */
44       bRes2 = false;
45       if
46       :: (i2 == i3) -> bRes2 = true;
47       :: else -> skip
48       fi;
49       if
50       :: bRes2 ->
51         /* translation of expression */
52         if
53         :: (v_request.stockID.intvalue ==
54           v_register.requestList.stockID[i1].intvalue)
55           -> bResult = true;
56         :: else -> skip
57         fi
58       :: else -> skip
59       fi;
60       /* update position() */
61       i2++;
62    :: else -> skip
63    fi;
64    i1++;
65 :: else -> break;
66 od;
```

Figure 4: XPath to Promela Example

Note that we have four boolean variables and five integer variables in the Promela translation. Boolean variable `bResult` is used to record the evaluation result of the whole XPath expression, `bRes1` and `bRes2` are used for evaluation of the two predicates on the right hand side of the expression, and `bRes3` is used during the evaluation of the `last()` function. Integer variables `i1` and `i4` are used as array indices in different parts of the Promela code, `i3` records the value of function call `last()`, and `i2` and `i5` are used for `position()` function.

It is not hard to see that we can compute the value of `last()` prior to the evaluation of the whole XPath expression, and we record its value in `i3`. The main body of the calculation is a loop searching for the proper value of array index `i4` which satisfies the first predicate (value of `stockID` greater than 5).

The main body to compute the whole boolean XPath expression is similar. There is a loop searching for the proper value of array index `i1`, and the code handling two predicates are nested. Note that position variable `i2` and array index `i1` are properly updated. According to the semantics of boolean expressions in the XPath standard, `bResult` is set to true once we find a value of `i1` satisfying the boolean expression.

Finally, note that there are more efficient Promela translations than the one presented in Fig. 4. For example, integer variable `i4` (for array index) can be reused to replace `i1`. In our implementation, we have a variable assignment optimizer to achieve this objective. However, we omit the details of its implementation here to simplify the presentation.

4.2 Supporting Data Structures

We can make the following observations based on the motivating example shown in Fig. 4: (1) Every XPath language construct (expression, path, step) corresponds to a Promela code segment. For example, the boolean XPath expression shown in Equation 2 corresponds to the whole Promela code in Fig. 4, its right hand side corresponds to the whole code with lines 51 to 57 left blank, and the left hand side corresponds to an "empty" statement since no code is generated for it. (2) In particular, loops are generated for those steps which generate XML data that corresponds to an MSL type with multiple occurrences (i.e., types declared as $g\{m, n\}$). For example, the step `stockID` in the right hand side corresponds to the loop from line 30 to line 66. (3) The generated code segments are embedded into each other. For example the segment (lines 34 to 63) that corresponds to "`[int()>5]`" is embedded in the code for step `stockID`; while it embeds the code for predicate "`[position()=last()]`" (lines 43 to 59). (4) The generated code can be regarded as an nested-loop which simulates the search procedures for each location path, and the evaluation of the boolean expression is placed in the body of the inner-most loop.

Our translation algorithm needs a mechanism to represent the structure of the input XML document and an approach to conveniently capture the Promela code segments that are generated and embedded into each other. Hence, we introduce two data structures which will be used in the translation algorithm: a *type tree* structure which represents the MSL types and a *statement macro* which represents (partially) generated Promela code segments. We also define several functions that manipulate these data structures.

Type Tree. We use the type trees to statically represent the input and output of an XPath location path (or a step). Given an XML variable and its MSL type, it is straightforward to derive the corresponding type tree. For example, Fig. 5 is the corresponding type tree for XML variable `register` in Equation 2, with the MSL type given in Example 2.3. Note that each node in the type tree corresponds to a subexpression of the MSL type expression given in Example 2.3, where the root node corresponds to the whole type expression. Hence, each node also corresponds to an MSL type.

In a type tree, each node is labeled with an MSL type (for the root node, it is also labeled with the XML variable name). Note that if the associated MSL type has multiple occurrence, the node is equipped with an additional index. For example, index `i1` is associated with node 5 in Fig. 5. Recall that an MSL type with multiple occurrence is translated into an array in Promela. This index is used to access the elements of that array. For each node in a

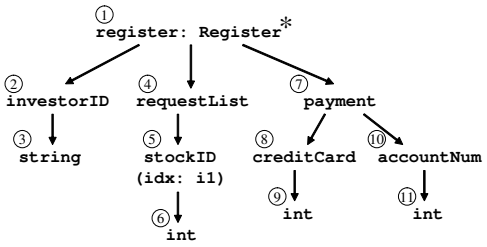


Figure 5: Type-Tree for Variable `register`

type tree, by tracing back to the root of the tree, we can get its *qualified name*, i.e., the expression in the Promela translation which accesses the data with the type represented by that node. For example, `v_register.requestList.stockID[i1].intvalue` is the qualified name of node 6, where the prefix `v_` is automatically added by the system to avoid name collision, and the `intvalue` is the name of the attribute with the basic type `int`.

We now define a number of functions on type trees. Given a type tree t, and an XPath step s, function MarkChild(t,s) proceeds as follows: (1) unmark all marked nodes in t, and (2) for each node that is unmarked in step 1, mark its children which are the results of executing step s, and (3) return the modified t. For example, let t_r be the type tree in Fig. 5 where node 1 is the only marked node (marked with "*"), let s be the step "`requestList`". The result of MarkChild(t_r,s) is the same type tree where node 4 is the only marked node. Other functions such as MarkParent(t), MarkAll(t), MarkRoot(t) work in a similar way. For example, nodes 6,9, and 11 are the marked nodes after executing MarkChild(MarkAll(t_r), "`int()`").

Statement Macro. In our translation algorithm, each XPath construct corresponds to a Promela code segment. A code segment can be regarded as a list of statement macros which are sequentially concatenated using "`;`". A statement macro (or simply macro) captures a block of Promela code for a certain functionality, and each macro has at most one *BLANK* space where another Promela code segment can be embedded. There are five types of macros we are using in our translation algorithm which are summarized below. A macro can have input parameters, and in the corresponding Promela code, the appearance of these parameters will be replaced by the actual input value when the macro is used.

Statement Macro	Promela Code

IF(v)	`if`
	`:: v -> BLANK`
	`:: else -> skip`
	`fi`

FOR(v,l,h)	`v = l - 1`
	`do`
	`:: v < h -> BLANK`
	`v ++`
	`:: else -> break`
	`od`

EMPTY	*BLANK*
INC(v)	`v ++`
INIT(v,a)	`v = a`

Fig. 6 presents a set of macros organized as a tree. There are two types of edges from a child to its parent: the embedment edge which is shown as a solid arrow and the sequential composition edge which is shown as a dotted arrow. We call such a tree a *macro tree*. In fact, any Promela code generated for an XPath expression construct can be captured using one or a set of macro trees. Given a macro tree, it is straightforward to generate the corresponding Promela code. For example, the macro tree in Fig. 6 corresponds to the code segment from line 30 to line 66 (with line 51 to line 57 as *BLANK*) in Fig. 4.

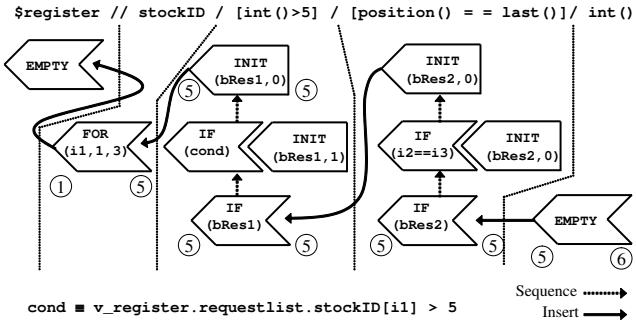


Figure 6: Code Sample

We associate two attributes with each macro: an input type node, and an output type node from a type tree. For each macro, the input node characterizes the starting node where the macro starts searching, and the output node is the starting point of its embedment. For example in Fig. 6, macro `FOR(i1,1,3)` is the corresponding code for the step "`//stockID`" in the location path at the top of the figure. Its input type node is the type node 1 in Fig. 5, which corresponds to the type of the XML variable `register`. Its output node (also the input node for its embedment) is the type node 5 (with MSL type `stockID`), which is the result of evaluating the step on the input type node. In our translation, except for concatenating code for two expressions, the input node of embedded code should match the output node of the *BLANK* where it is inserted.

We also associate a hashtable with each macro, which records the mapping from XPath location paths to qualified names. For example, the hashtable of the last EMPTY macro in Fig. 6 will map the location path shown in Fig. 6 (i.e., right hand side of Equation 2) to qualified name of node 6, i.e., the expression "`v_register.stockID[i1].intvalue`".

We have three different functions to embed macros into each another: MatchInsert(c_1,c_2), InsertAll(c_1,c_2), and InsertAndReplace(c_1,c_2). All these functions return one macro tree that is the result of embedding c_2 into the *BLANK*s of c_1. In our translation, c_1 is always guaranteed to be a single macro tree, while c_2 can be a set of macro trees. The MatchInsert function requires that the inserted macro tree must match the output node of its host; while InsertAll and InsertAndReplace do not require matching. When inserting c_2 into the *BLANK*s of c_1 InsertAndReplace replaces the location paths in c_2 with qualified names based on the hashtable of the host.

Function GenCode(n) generates a macro tree given a type node n. GenCode relies on a global registry R which registers index variables that are processed before. The function traces back from the current type node to the root type node. Whenever an unprocessed index is encountered, a FOR macro is generated for that index, and the index is registered in R. When a new FOR macro is generated, the old FOR macro generated before is embedded in the new macro to form a nested loop. For example, if index `i1` has not been processed, GenCode(n_6) generates a FOR(`i1`,1,3), given n_6 is the node 6 in Fig. 5.

(1) $exp \rightarrow exp_1$ `op` exp_2:
exp_1.*inTree* = exp.*inTree*
exp_2.*inTree* = exp.*inTree*
If exp is intermediate **Then**
exp.*code* = InsertAll(exp_1.*code*,exp_2.*code*)
else
exp.*code* =
InsertAndReplace(InsertAll(exp_1.*code*, exp_2.*code*),IF(exp))
where the *BLANK* of IF(exp) is filled with "*exp.var* = `true`"
End If

(2) $exp \rightarrow$ `op` exp_1:
exp_1.*inTree* = exp.*inTree*
exp.*code* = exp_1.*code*

(3) $exp \rightarrow$ `const` :
exp.*code* = EMPTY

(4) $exp \rightarrow p$:
p.inTree = exp.*inTree*
exp.*code* = *p.code*

(5) $p \rightarrow$ `$v` p_1:
p_1.*inTree* = generate a type tree for `$v`
p.outTree = p_1.*outTree*
p.code = p_1.*code*

(6) $p \rightarrow$ / p_1 | // p_1:
If $p \rightarrow /p_1$ **Then**
p_1.*inTree* = MarkRoot(*p.inTree*)
Else
p_1.*inTree* = MarkAll(*p.inTree*)
End If
p.code = p_1.*code*
p.outTree = p_1.*outTree*

(7) $p \rightarrow p_1$ / s | p_1 // s:
p_1.*inTree* = *p.inTree*
If $p \rightarrow p_1 / s$ **Then**
s.*inTree* = MarkRoot(p_1.*outTree*)
Else
s.*inTree* = MarkAll(p_1.*outTree*)
End If
p.code = MatchInsert(p_1.*code*,s.*code*)
p.outTree = s.*outTree*

(8) $s \rightarrow$. :
s.*outTree* = s.*inTree*
s.*code* = EMPTY

(9) $s \rightarrow$ .. :
s.*outTree* = MarkParent(s.*inTree*)
s.*code* = EMPTY

(10) $s \rightarrow b() \mid t \mid *$:
s.*outTree* = MarkChild(s.*inTree*,s)
s.*code* = $\{c_1, \ldots, c_k\}$ where
c_i is GenCode(d_i) for each type node d_i in s.*outTree*

(11) $s \rightarrow [exp]$:
exp.*var* = a unique variable name
s.*outTree* = s.*inTree*
s.*code* = exp.*code* ";" IF(exp.var)

Figure 7: Translation Algorithm

4.3 Syntax Directed Translation Algorithm

Now we discuss the syntax directed translation algorithm which is presented in Fig. 7. Each non-terminal (e.g. exp, p, s) has one inherited-attribute: *inTree*, and two synthesized-attributes: *outTree* and *code*. Attributes *inTree* and *outTree* are both type trees, and they are used to capture the input and output of XPath language constructs, respectively. Attribute *code* is a set of macro trees, which records the generated Promela code that corresponds to the non-terminal. Non-terminal exp has an additional attribute *var*, which is the boolean variable that records the evaluation results for the exp. The *var* attribute is not null if and only if exp is not an intermediate expression (e.g. exp is used as a branch condition in host language or it is a predicate in another XPath location path). For example, when generating the Promela code (Fig. 4) for Equation 2, the attribute `var` for the boolean expressions in predicates "`[int()>5]`" and "`[position()=last()]`" are `bRes1` and `bRes2` respectively.

Handling of Expressions. Rules 1, 2, 3, and 4 handle the translation of XPath (boolean or arithmetic) expressions. In rule 1 both subexpressions inherit the *inTree* from exp. For example, when evaluating Equation 2 the *inTree* to inherit is null. For another example, when processing the expression "`int() > 5`", the *inTree* to inherit is a version of the type tree shown in Fig. 5 where the node 5 is the only marked node. We will discuss where instances of type trees are generated later in the handling of XPath location paths.

The *code* of exp is synthesized from the *code* of the two subexpressions. The basic idea is to embed the code generated by exp_2 into the code of exp_1, regardless of the matching of input/output type node (by the use of InsertAll instead of MatchInsert). If exp is not intermediate (e.g. it is used as a boolean branching condition), we need additional processing (calling InsertAndReplace) to insert another IF macro into the synthesized code. The IF macro assigns `true` to attribute *var* if the exp evaluates to true. Hence the generated code evaluates the boolean expression and stores the result in *var*. For example, as we mentioned earlier, the code for the right hand side of Equation 2 is the whole code in Fig. 4 except lines 51 to 57 are *BLANK*, and the code for the left hand side is an EMPTY macro. When we synthesize the *code* for Equation 2 from these two subexpressions, an IF macro (which assigns the *var*, i.e., the `bResult`) is embedded in that *BLANK* (lines 51 to 57) by the call to InsertAndReplace. Note that the location paths of Equation 2 are replaced by qualified names.

The rest of the expression related syntax rules, i.e., rules 2, 3, and 4, work in a similar way: they pass down information *inTree* to subexpressions, and synthesize *code* and *outTree* from subexpressions. Finally, note that *outTree* is not used in the syntax rules for expressions.

Handling of Location Paths. Rules 5, 6, and 7 handle the translation of an XPath location paths. In rule 5, where a location path is associated with an XML variable, a corresponding type tree is generated and passed to the steps of the path. For example, to handle the right hand side of Equation 2, the type tree in Fig. 5 is generated. Note that even for the same XML variable, when a new type tree instance is generated, the index attributes should have unique names. For example, when pre-calculating the value of `last()` (line 11 to 28), another type tree is generated for XML variable `register`, and the index of node 5 is `i4` (instead of the `i1` in Fig. 5).

Rule 6 handles the absolute location paths, where the inherited attribute *inTree* is handled differently for XPath operator "`/`" and "`//`" respectively. Rule 7 processes a path, step by step and from left to right, as it passes the *outTree* of the partial path p_1 to the step s on the right. Note that when synthesizing *code*, we need to match the type node when embedding macros, so MatchInsert is called.

Handling of Steps. Rules 8, 9, 10, 11 handle steps. The semantics of rules 8 and 9 is clear. Rule 11 calls MarkChild function to symbolically execute the step s on the *inTree*. For each type node d_i in the *outTree*, function GenCode is called to generate a macro tree for d_i. Finally, rule 11 handles the case when the step is a predicate,

for example, the boolean expression "`int() > 5`" (let us call it e_2) in Equation 2. Its synthesized code consists of two parts: an evaluation code for the expression (lines 35 to 40 for the evaluation of e_2), and an IF macro (lines 41 to 63) which allows insertion of code for later steps (lines 44 to 61).

4.4 Handling of Function Calls

The handling of `position()` and `last()` calls is a little bit more complicated, though the idea is similar: substitute the appearance of a function call with an integer variable, and properly update its value so that when the function is called the integer variable contains the right value.

Each `position()` (or `last()`) call has an *owner* which is an non-intermediate boolean XPath expression where the call appears. For an owner exp, we need another attribute called *prefix* which contains Promela macros (just like the *code* attribute). The code in *prefix* will be placed ahead of the code contained in *code* to form the complete code for the owner.

When a `position()` call is encountered, we acquire a unique integer variable for that call (let it be v). Then we append the macro INIT(v,1) in the *prefix* attribute of *owner* and insert the INC(v) in the *BLANK* of the macros generated by the immediate previous step. For example, to handle the `position()` of Equation 2, the integer variable `i2` is acquired, and its initialization statement is at line 8, and its update statement is at line 61 (which is inside the *BLANK* of the code that corresponds to the previous step "`[int()>5]`").

The handling of `last()` is even more complicated: it works in three modes: normal mode, copy mode, and processed mode. The normal mode is for the first time the `last()` call is encountered; in the copy mode the `last()` is encountered for a second time when the pre-calculation code is being generated; the processed mode is the case where the value for the `last()` call has been pre-calculated and this value should not be changed any more. Consider the `last()` call in Equation 2 as an example. In the normal mode, we acquire an integer variable for the `last()` call (i.e., `i3`), and call the handling of its owner (i.e., Equation 2) to pre-calculate the value of `last()` (hence line 9 to line 28 will be generated). Now when the second translation of Equation 2 reaches the `last()` call, it is in the copy mode. The initialization and update statements are generated for the pre-calculation code (i.e., line 10 and line 22). When we return from the pre-calculation, the processing of the `last()` enters the processed mode, and `i3` is not allowed to be changed. The appearance of `last()` in the second predicate is replaced by `i3`.

EXAMPLE 4.1. The translation of Equation 2 is split into two recursive translation tasks on its left and right hand paths. It is not hard to see that the left hand side generates an EMPTY macro. Now we concentrate on the right hand side, which is converted to the following form:

```
$register//stockID/[int()>5]/[position()=last()]/int()
```

The translation algorithm will start from `$register` and then processes steps from left to right. First a type tree for `register` (as shown in Fig. 5) is generated. Then function MarkAll is called, and the resulting tree is passed as the *outTree* to step `stockID`. In the *outTree* of step `stockID`, node 5 will be the only marked node. Then function GenCode is called for node 5, which generates a FOR macro that corresponds to lines 31 to 65 in Fig. 4 (with lines 34 to 63 as *BLANK*). The handling of the next step `[int()>5]` is similar, where an IF macro is generated and embedded into the FOR macro generated before. For the third step `[position()=last()]`, integer variables `i2` and `i3` are acquired for the function two calls respectively. The initialization and update statements for `position()` are generated (line 8 and line 61). However since it is in the normal mode for `last()`, we do not generate any code for `last()`. Instead, the translation of Equation 2 is called again for the pre-calculation of `last()`, and lines 9 to 28 are generated. After the return from the second translation call on Equation 2, the first translation call advances to the last step `int` which generates an EMPTY macro whose hashtable contains the information that maps the right hand side location path to the corresponding qualified name. Finally, when synthesizing the *code* attribute for Equation 2, an IF macro (lines 51 to 57) is inserted and the two location paths in Equation 2 are replaced with qualified names. ∎

5. APPLICATIONS

In this section we discuss the applications of our techniques to the verification of web services. We present a case study, where our techniques help to identify a very delicate design error of XPath expressions in a conversation protocol. Then, we briefly discuss our work on verifying interacting BPEL4WS web services, and the Web Service Analysis Tool (WSAT), where the techniques presented in this paper constitute the basis.

A conversation protocol [3, 7, 9] is a top-down specification, which specifies desired global behaviors (message sequences) of a composite web service. Formally, a *conversation protocol* is a tuple $\langle (P, M), \mathcal{A} \rangle$ where the composition schema (P, M) defines the set of peers (participants of the composite web service), and the message classes that are transmitted among peers. $\mathcal{A}$ is a Guarded Finite State Automaton (GFSA) $\mathcal{A} = (M, T, s, F, \Delta)$, where M is the set of message classes as defined in the composition schema (here a message class consists of an MSL type, the sender and receiver), T is a finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states, and Δ is the transition relation. Each transition $\tau \in \Delta$ is of the form $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of τ, $c \in M$ is a message class and g is the *guard* of the transition. A guard consists of a guard condition and a set of assignments. A transition is taken only if the guard condition evaluates to true. The assignments specify the contents of the message that is being sent. Given a transition $\tau = (s, (c, g), t)$ where peer p is the sender of the message of type c, then guard g is a predicate of the following form: $g(m, \vec{m})$, where m is the message being sent, and the vector $\vec{m}$ contains the last instance of each message type that is received or sent by peer p. Guards are written using XPath expressions.

We now present a conversation protocol named Stock Analysis Service (SAS). Fig. 8 presents its overall structure and control flow, and Fig. 9 is a fragment of its formal specification. As shown in Fig. 8, SAS involves three peers: Investor (Inv), Stock Broker Firm (SB), and Research Department (RD). Inv initiates the stock analysis service by sending a `register` message to SB. SB may `accept` or `reject` the registration. If the registration is accepted, SB sends an analysis `request` to RD. RD sends the results of the analysis directly to Inv as a `report`. After receiving a `report`, Inv can either send an `ack` to SB or `cancel` the service. Then, SB either sends the `bill` for the services to Inv, or continues the service with another analysis `request`.

In Fig. 9 we present a partial specification of the SAS protocol. The specification of SAS consists of two parts: a schema and a GFSA protocol. The schema specifies the set of peers, a list of MSL types, and a list of peer to peer message classes which are built upon the MSL types. The GFSA specification consists of states, and transitions. We present two key transitions from the protocol: `t8` and

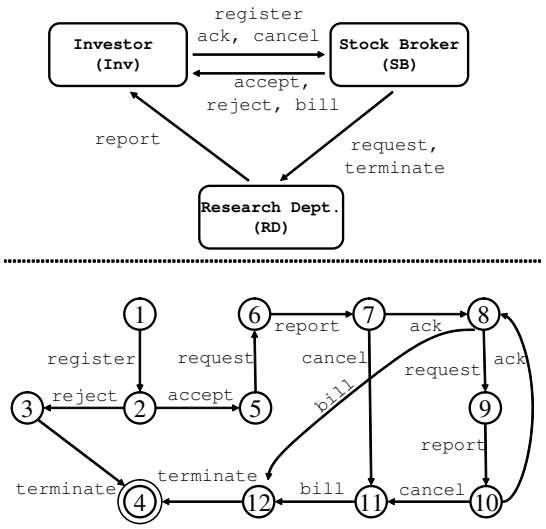


Figure 8: Stock Analysis Service

t14. A transition is equipped with a guard which determines the transition condition and the assignment of the message being sent. For example, transition t8 sends a message of type request. Its transition condition is the following boolean XPath expression:

```
$request//stockID/int() !=
  $register//stockID [position() = last()]/int()
```

The rest of the guard assigns values to the investorID and the stockID fields of the request message being sent. According to the semantics of GFSA [8], except the request message which appears at the left side of assignment operator ":=", the appearance of all other message classes refers to the *latest* copy of that message class. Hence the transition condition of t8 means "if the stockID of the latest request message is not the last stockID of register message". Its assignment tries to send the stockID which is subsequent (in the register message) to the stockID appeared in the latest request message. Similarly the guard of transition t14 specifies that if the latest request message contains the last stockID in the register message, then a bill message is sent to conclude the interaction. Generally the logic of t8 and t14 intends to send out the list of stockID in the initial register message one by one.

It is not hard to see that the SAS protocol can be translated into a Promela process (the translation of XPath and MSL to Promela is presented in this paper, and the translation of the control flow of a GFSA is discussed in [8]). Note that in the Promela translation, there is an initialization stage which assigns initial values to all messages nondeterministically.

Given the logic of the transitions t8 and t14, it is natural to propose the following LTL property for the Promela translation of the SAS protocol:

```
G (
   (
    index < v_register.requestList.stockID_occ &&
    v_register.requestList.stockID[index].intvalue == value
    && msg == m_register
   )
   ⇒
   (
    F(msg == m_reject) ||
    F(msg == m_cancel) ||
    F(request.stockID == value)
   )
)
```

```
Conversation {
 Schema{
  PeerList{Inv,SB,RD},
  TypeList{
   Register[
    investorID[xsd:string],
    requestList[
     stockID[xsd:int]{1,3}
   ],
   payment [
     accountNum[xsd:int] |
     creditCard[xsd:int]
   ],
  },
  ...
  MessageList{
   register{Inv -> SB: Register},
   reject{SB -> Inv: Reject},
   ...
  }
 },

 Protocol{
  States{s1,s2,...,s12},
  InitialState {s1},
  FinalStates{s4},
  TransitionRelation{
   ...
  t8{s8 -> s9 : request,
    Guard{
     $request//stockID/int() !=
       $register//stockID [position() = last()]/int() =>
     $request[
      //investorID := $register//investorID,
      //stockID :=
       $register // stockID
         [ position() = $register // stockID
             [int()=$request//stockID/int()]/position()+1
         ]
     }
  },
  t14{ s8 -> s12 : bill,
     Guard{
      $request//stockID =
         $register//stockID [position() = last()] =>
      $bill[
            //orderID:= $register//orderID
          ]
     }
  },
   ...
  }
 }
}
```

Figure 9: A Sample Conversation Specification

In the above LTL property, temporal operator **G** means "globally", temporal operator **F** means "eventually" and *index* and *value* are two predefined constants. The variables starting with v_ are the qualified names referring to XML data, as we have discussed in Sections 3 and 4. The variable msg is a variable in the Promela translation for GFSA, which records the current message being sent. For example, when transition t8 is executed, msg will be assigned the value m_request.

The LTL property states that: if the register message contains a stockID (at position *index*, with value *value*), then eventually there should be a request containing that stockID, if nothing wrong happens (i.e., the register is not rejected, and the Inv does not cancel the service).

Interestingly, SPIN soon identifies that the SAS specification does not satisfy the proposed LTL property. SPIN gives an error-trace where the register message has three stockIDs with values 0, 1, 0 respectively. The error-trace shows that when the first request for stockID 0 is sent, transition t8 is disabled because the stockID of the latest request is the last stockID in the register message; instead, the transition t14 is triggered

to send out the `bill` message to conclude the interaction. The verification identifies the error in the design of XPath transition guards which rely on the presumption that "there should be no redundant `stockID`s in the `register` message", however this is not enforced by the specification.

As SPIN is an explicit model checker, the verification, unfortunately does not scale very well. When integer domain is set to [0,1], the verification time is 3 seconds and memory consumption is around 50MB. When the domain is increased to [0,3], the memory consumption grows to over 600MB. However, our experience shows that SPIN is still useful in identifying errors in protocols by restricting the data domains.

Complexity of data manipulation is only one of the challenges that arise in model checking composite web services. Since communication among web services is asynchronous, most interesting problems in analyzing web services become undecidable [7]. In [7, 8], we proposed several sufficient conditions which restrict control flows of web services so that undecidability induced by the asynchronous communication can be avoided in the verification process. Decision procedures for these sufficient conditions are implemented in our research project WSAT [10, 19]. WSAT also supports LTL model checking for composite web services that are specified in popular industry standards such as BPEL4WS. The key idea is to translate BPEL4WS services into GFSA representation [8], and then the techniques presented in this paper can be applied to translate from GFSA to Promela. The WSAT can be extended in the future to support more web service specification languages (such as DAML-S, WSCI, etc.).

6. CONCLUSION

In this paper we presented techniques for representing XML data and XPath expressions in Promela. These techniques allow us to verify LTL properties of XML manipulating software, such as web services. In contrast to earlier work, our approach does not abstract away the XML data manipulation. We implemented the algorithms in our WSAT tool, and verified several example systems including conversation protocols and BPEL4WS web services. As future work we are planning to apply symbolic model checking techniques to tackle the large state spaces caused by XML data.

Acknowledgments

Bultan was supported in part by NSF Career award CCR-9984822 and NSF grant CCR-0341365; Fu was partially supported by NSF Career award CCR-9984822, NSF grants IIS-0101134 and CCR-0341365; Su was supported in part by NSF grants IIS-0101134 and IIS-9817432.

7. REFERENCES

[1] Business Process Execution Language for Web Services (BPEL4WS), version 1.1. *available at* `http://www.ibm.com/developerworks/library/ws-bpel`.

[2] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL a model for W3C XML Schema. In *Proc. of 10th World Wide Web Conf. (WWW)*, pages 191–200, 2001.

[3] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. of the 12th Int. World Wide Web Conf. (WWW)*, pages 403–410, May 2003.

[4] J. C.Corbett, M. B.Dwyer, J. Hatcliff, S. Laubach, C. S. Pasarenau, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE)*, pages 439–448, 2000.

[5] A. Simon Christensen, A. Møler, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. 18th IEEE Int. Conf. on Automated Software Engineering (ASE)*, 2003.

[7] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. 8th Int. Conf. on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 188–200, 2003.

[8] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL Web Services. To appear in the *Proc. of 13th Int. World Wide Web Conf. (WWW)*, 2004.

[9] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. To appear in the *Proc. of 2004 IEEE Int. Conf. on Web Services (ICWS)*, 2004.

[10] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web service compositions. To appear in the *Proc. of 16th Int. Conf. on Computer Aided Verification (CAV)*, 2004.

[11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[12] M. Koshkina and F. van Breugel. Verification of business processes for web services. Technical report, York University, 2003.

[13] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. of 21th Symposium on Principles of Database Systems (PODS)*, pages 65–76, 2002.

[14] A. Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

[15] S. Narayanan and S. A. Mcllraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th Int. World Wide Web Conf.*, 2002.

[16] F. Neven. Automata theory for XML researchers. *Sigmod Record*, 31(3), 2002.

[17] The OWL Services Coalition. OWL-S: Semantic markup for web services. `http://www.daml.org/services/owl-s/1.0/owl-s.pdf`, 2003.

[18] W3C. Web Services Description Language (WSDL) version 1.1. *available at* `http://www.w3.org/TR/wsdl`.

[19] Web Service Analysis Tool (WSAT). `http://www.cs.ucsb.edu/~su/WSAT`.

[20] Web Service Choreography Interface (WSCI). `http://www.w3.org/TR/wsci/`.

[21] Extensible markup language (XML). *available at* http://www.w3c.org/XML.

[22] XML Path Language. *available at* `http://www.w3.org/TR/xpath`.

[23] XML Schema. *available at* `http://www.w3c.org/XML/Schema`.