XSLT and Automata with Equality Constraints

Aleksy Schubert and Daria Wałkiewicz-Chrząszcz

Institute of Informatics
Warsaw University
ul. Banacha 2
02–097 Warsaw
alx@mimuw.edu.pl

October 28, 2007

Abstract

This paper presents a new kind of automata — automata with equality constraints on coordinates. These automata allow to control the operation of the XSLT transformations and allow to check equality specifications in addition to traditional XML Schema based consistency. This enables the possibility for additional XSLT transformations sanity checks that ensure equality of data copied from the source document to the resulting one.


1 Introduction

XML is an established format for structured data. It allows to formulate a wide range of formats to represent information for a wide range of applications. An XML language is a subset of XML documents typically described by formalisms known as schemas. Many different schemas have been proposed: the original DTD mechanism (that is a part of the XML specification [W3C06]), XML Schema [W3C04a, W3C04b], RELAX NG [CM01], DSD2 [Møl02] and various tree formalisms [HP03, W3C07]

XSLT [W3C99] is a language which allows to write transformations of the XML documents [W3C06]. The XSLT stylesheets serve to transform information encoded in one kind of documents into documents of another kind. Usually, the structured information from a database should be made available in a format which makes possible an elegant presentation (in most cases as an HTML file). Another use of XSLT is to change the format of data in order to be comprehensive for other applications. This includes filtering, copying and deleting some data.

Each XSLT transformation consists of a set of rules. Each of the rule has two parts: a pattern which describes the points where the rule is to be applied in the source document and a template which describes the content of a fragment of

This work was partly supported by Polish government grant 3 T11C 002 27.
the resulting document. The processing starts with the rule which is applicable to the root of the document.

Someone who develops a particular XSLT transformation may be interested if the transformation is correct. The simplest aspect of correctness is a syntactic check. Another one consist in verifying that a transformation uses elements that exist according to the schema. In practice, one wants also to have a way to check if the transformation makes sense. One of the possible approaches is to check if given a particular kind of the input documents the transformation gives desirable outputs. This can be formalised in the following algorithmic problem introduced in [MSV00]:

**Problem 1 (checking of XSLT sheets)**

Input: a type $\tau_1$ of the input documents, a type $\tau_2$ of the output documents and a stylesheet $t$.

Question: Does the stylesheet $t$ generate a document of the type $\tau_2$ when given a document of the type $\tau_1$?

One of the possible ways to approach this problem is to employ finite tree automata to check these properties. This approach is taken by Milo et al in [MSV00]. A more refined notion of transducer is used to model the fact that the transformations change the input documents into output ones. Moreover, Milo et al consider the top-down automata as they give rise to more natural translation of DTDs to automata.

In this paper we propose a slightly different approach and use the traditional bottom-up automata, but on product trees and with equality constraints between the coordinates. The first coordinate corresponds to an input specification, the second to the output one. Equality constraints allow to express that certain parts of the input should be left unchanged or should be copied without changes.

This additional device allows to capture desirable properties of the XML transformations involving copying, deletion and filtering. Let us imagine a transformation (inspired by the addrbok example from [HP03]), which creates a telephone book document from an address book document by keeping only these entries that have telephone address. Or a transformation that given a document of books generates a table of contexts for each book, possibly followed by a summary of the book, given by listing the title and the introduction of each chapter (example presented in [MN07]).

Using equality test we can also more easily control updates, restricting the ways in which data is modified. To illustrate this problem consider an example from [CAM07] where an XML document is exchanged between three parties, Source, Broker and User. Assume Broker is allowed to modify data he receives from Source but only in a limited way, for example by adding advertisements in certain well-defined areas. Equality tests in the automaton corresponding to the transformation will ensure that the data from Source is not altered. In general, equality tests seem to be especially important for the almost-identity transformations, or transformations that keep some parts of the input unchanged, which in practice happens quite often.

2
2 Notation

This section recalls preliminary notions used in the rest of the paper. It serves to settle the notation.

We use labelled trees extensively. The trees are usually denoted by small Latin letters like t, s, u, etc. The labels belong to signatures which are usually denoted by \( \mathcal{F}, \mathcal{F}' \), etc. Each signature is equipped with the arity of the labels \( ar : \mathcal{F} \rightarrow \mathbb{N} \). The set of all finite trees over a signature \( \mathcal{F} \) is denoted by \( T(\mathcal{F}) \).

In order to address positions in a tree we use sequences of natural numbers. These sequences are denoted by small Greek letters like \( \gamma, \rho \), etc. The empty sequence is denoted by \( \varepsilon \). The subtree of \( t \) at the address \( \gamma \) is denoted by \( t|_{\gamma} \).

We compose addresses so that \( f(t_1, t_2)|_{1\cdot\gamma} = t_1|_{\gamma} \). The result of replacement of the subtree at position \( \gamma \) in \( t \) with the tree \( s \) is denoted by \( t[\gamma \leftarrow s] \).

The positions are compared by the prefix order which is denoted by \( \preceq \). For \( \gamma \preceq \rho \), we define \( \rho \setminus \gamma \) as the suffix of \( \rho \) that starts after \( \gamma \). The length of a path \( \gamma \) is denoted by \( |\gamma| \). The set of all addresses that are applicable to a tree \( t \) is denoted by \( \text{Pos}(t) \). The positions are also compared by the lexicographic order which is denoted by \( \leq \).

3 Automata on products of trees

This section presents the notion of automata with constraints on product trees.

3.1 Product trees

A definition of an automaton with constraints on product trees requires the notion of a product tree. This notion is introduced here. The presentation of the product alphabet is based on the one in [CDG+97]. Let \( \mathcal{F} \) be a finite alphabet of first-order function symbols. Let \( ar : \mathcal{F} \rightarrow \mathbb{N} \) be a function which for a given symbol \( f \in \mathcal{F} \) returns its arity. We define the \( n \)-ary product alphabet, denoted \( \mathcal{F}^n \), as the set \( \hat{\mathcal{F}}^n \) where \( \hat{\mathcal{F}} = \mathcal{F} \cup \{ \bot_a \} \) with arity

\[
ar(f_1, \ldots, f_n) = \max(ar'(f_1), \ldots, ar'(f_n))\]

where \( ar' \) is equal to the original \( ar \) except for \( ar'(\bot_a) = 0 \). By convention, we write \( f_1 \ldots f_n \) instead of \( (f_1, \ldots, f_n) \) to denote a symbol which is a tuple of elements \( f_1, \ldots, f_n \).

We define an operator which transforms a tuple of terms over \( \mathcal{F} \) into a term over \( \mathcal{F}^n \) as follows

\[
[f_1(t_1^1, \ldots, t_k_1^1), \ldots, f_n(t_1^n, \ldots, t_k^n)] = f_1 \ldots f_n([t_1^1, \ldots, t_k_1^n], \ldots, [t_1^n, \ldots, t_k^n])
\]

where \( t_j^i = \bot_a \) for \( j > k_i \). We define also operators \( \pi_i \) for \( i = 1, \ldots, n \) that project a tuple term into a specified \( i \)-th coordinate.

\[
\pi_i(f_1 \ldots f_n(t_1, \ldots, t_n)) = \begin{cases} 
\bot_a & \text{if } f_i = \bot_a \\
 f_i(\pi_i(t_1), \ldots, \pi_i(t_{k_i})) & \text{otherwise}
\end{cases}
\]

where \( k_i = a(f_i) \). Note that the operation does not need to give a result in the term algebra over \( \mathcal{F} \). The resulting term may contain occurrences of \( \bot_a \).
This kind of tree allows to define a new form of position. A fine-grained position in a tree $t$ is denoted by $\pi_i\gamma$ where $\pi_i$ is one of the applicable projections and $\gamma$ is a position in $t$. For such a position we define $t|_{\pi_i\gamma} = \pi_i(t)|_{\gamma}$. We omit the adjective fine-grained whenever there is no ambiguity.

**Definition 2** (order $\leq$ on fine-grained positions)
We extend the order $\leq$ on positions to fine-grained positions so that $\pi_i\gamma \leq \pi_j\gamma'$ if $\gamma < \gamma'$ or $\gamma = \gamma'$ and $i \leq j$.

The presentation above is restricted to products of a single alphabet. Clearly, it can be easily generalised to any product as well as the further part of the paper.

### 3.2 Automata with constraints

We present a new notion of an automaton on trees. The automata introduced here are augmented with equality constraints that compare subtrees on different coordinates. The presentation of the automata is loosely based on [CDG+97, CCC+94].

**Definition 3** (equality constraints)
A product positive atomic equality constraint is an expression of the form $\pi_i\gamma \doteq \pi_j\gamma'$ where $\gamma, \gamma' \in \{1, \ldots, k\}^*$ and $k$ is the maximal arity of symbols in $\mathcal{F}$. We say that such an expression is defined on a term $t$ when

- both $\pi_i(t)$ and $\pi_j(t)$ are defined,
- $\gamma \in \text{Pos}(\pi_i(t))$ and $\gamma' \in \text{Pos}(\pi_j(t))$.

Such a predicate is satisfied on a term $t$, which is written $t \models \pi_i\gamma \doteq \pi_j\gamma'$, when it is defined and $\pi_i(t)|_{\gamma} = \pi_j(t)|_{\gamma'}$.

The satisfaction relation is extended as usual to conjunctions of any product positive atomic equality constraints. These extended constraints together with atomic ones are called here, for brevity, constraints. A conjunction of zero constraints is denoted by $tt$.

We treat $\doteq$ as a symmetric operation so in definitions and formulations of facts we omit cases symmetric wrt. the equality.

**Definition 4** (automata with product constraints)
An automaton with product constraints or simply automaton is a tuple

$$A = (Q, T, \omega, \mathcal{F}^n, Q_f, \Delta),$$

where $Q$ is a finite set of states, $T$ is a finite lattice, $\omega : Q \rightarrow T$ is a function, $\mathcal{F}^n$ is a product of a finite alphabet, $Q_f \subseteq Q$ is the set of final states and $\Delta$ is a set of transition rules of the form

$$f(q_1, \ldots, q_m) \xrightarrow{c} q$$

where $f \in \mathcal{F}^n$, $m = \text{ar}(f)$, $q_1, \ldots, q_m, q \in Q$, and $c$ is a constraint. We need a further restriction that $\omega(q) \leq \omega(q_i)$ for each $i = 1, \ldots, m$ and, additionally, if $c$ contains equalities then $\omega(q) < \omega(q_i)$ for each $i = 1, \ldots, m$. 

4
The definition is based on the definition of reduction automata from [CCC+94]. However, the reduction automata have the emptiness problem decidable, this is not the case for the automata introduced here already in case when one occurrence of the equality constraint is admitted [Tre00].

The above-mentioned notion of automaton is very strong and requires complicated arguments to prove its properties. We use a restricted version of the automaton which largely simplifies the development.

**Definition 5 (substitution automata)**

A substitution automaton \( A = (Q, T, \omega, F^n, Q_f, \Delta) \) is an automaton with product constraints on which additional restrictions are imposed:

- \( T = \{0, 1\} \) with the order \( 0 < 1 \),
- for each constraint \( c \) and each pair of atomic constraints
  \[
  \pi_{i_1} \gamma_1 \equiv \pi_{i_2} \gamma'_1, \quad \pi_{i_2} \gamma_2 \equiv \pi_{j_2} \gamma'_2
  \]
  in \( c \) if \( i_1 = i_2 \) and \( \gamma_1 \preceq \gamma_2 \) then \( \gamma_1 = \gamma_2 \).

**Definition 6 (basic functions on rules)**

Let \( A = (Q, T, \omega, F^n, Q_f, \Delta) \) be an automaton. Let \( r : f(q_1, \ldots, q_n) \mapsto q \) be a rule in \( \Delta \). We define \( \text{label}(r) = f, \text{state}(r) = q, \text{constr}(r) = c \).

For two constraints \( c = c_1 \land \cdots \land c_m \) and \( c' = c'_1 \land \cdots \land c'_m' \) we write \( c \equiv c' \) when \( \{c_1, \ldots, c_m\} = \{c'_1, \ldots, c'_m'\} \).

### 3.3 \( A \)-runs

We introduce the notion of a calculation for automata we introduced.

**Definition 7 (\( A \)-runs)**

Let \( A = (Q, T, \omega, F^n, Q_f, \Delta) \) be an automaton with product constraints. We define recursively the set of \( A \)-runs and a function \( \text{last} \) together with a function \( \text{term} \), both of which take runs as arguments:

1. For each zero-ary constant \( u \in F^n \) and \( (u \mapsto c) \in \Delta \), a tree \( \mathcal{R} \) that has a single vertex labelled with \((u \mapsto q)\) is a run. Additionally, \( \text{last}(\mathcal{R}) = q \) and \( \text{term}(\mathcal{R}) = u \).

2. Let \( \mathcal{R}_i \) be \( A \)-runs such that \( \text{term}(\mathcal{R}_i) = u_i \) and \( \text{last}(\mathcal{R}_i) = q_i \) for \( i = 1, \ldots, l \). Let \( (f(q_1, \ldots, q_l) \mapsto c) \in \Delta \) then
   
   (a) a tree \( \mathcal{R} \)
   
   i. the root of which is labelled with \((f(q_1, \ldots, q_l) \mapsto q)\),
   
   ii. has \( l \) sons such that the subtree at the \( i \)-th son is \( \mathcal{R}_i \) for \( i = 1, \ldots, l \), and
   
   iii. such that \( f(u_1, \ldots, u_l) |= c \)
   
   is an \( A \)-run,

   (b) \( \text{last}(\mathcal{R}) = q \),

   (c) \( \text{term}(\mathcal{R}) = u \).
We say that the run $R$ is on the tree $t$ when $\text{term}(R) = t$.

Note that each run is a tree so we can use positions in runs as positions in usual trees. In proofs, we sometimes loosen the notion above and drop requirements connected with states or constraints. In this case, the resulting objects are called simply trees. We slightly modify the notion of fine-grained position. Fine-grained positions are used for a run $R$ as if they were applied to $\text{term}(R)$.

We extend the operations from Definition 6.

**Definition 8 (basic functions on runs)**

Let $A$ be an automaton. Let $\gamma$ be a position in an $A$-run $R$. The expression $R(\gamma)$ denotes the label at the root of $R | \gamma$. Let $r$ be the label at the root of $\gamma$. We define $\text{state}_R(\gamma) = \text{state}(r)$ together with $\text{constr}_R(\gamma) = \text{constr}(r)$, $\text{label}_R(\gamma) = \text{label}(r)$, and $\text{term}_R(\gamma) = \text{term}(r)$.

Note that $\text{state}_R(\gamma) = \text{last}(R | \gamma)$.

**Definition 9 (language)**

The language accepted (recognised) by an automaton $A$ is defined as $L(A) = \{ u \mid u = \text{term}(R) \text{ for some } A\text{-run } R \text{ with } \text{state}(R) \in Q_f \}$.

**Definition 10 (the emptiness problem)**

*Input:* an automaton $A$ with equality constraints.

*Question:* Is there an accepting run of the automaton?

---

4 Automata with constraints and XSLT

In this section we sketch the way automata with constraints can be used to capture interesting properties of the XSLT transformations. We consider here a transformation $T$ between an XML application $A$ and an XML application $B$ (see Figure 1).

Consider the simple XMLSchema specification of the document format for the Application $A$ which is presented on Figure 2. In this document, the specification

```xml
<xs:element name="person" type="personType"/>
```
says that the name of the main element in the document for the application A is \( \text{⟨person⟩} \). Furthermore, the content of the element is defined by the type called \( \text{personType} \). The further examination of the XMLSchema document reveals that the elements of \( \text{personType} \) must have four children, in sequence: \( \text{⟨firstname⟩} \), \( \text{⟨familyname⟩} \), \( \text{⟨blob⟩} \) and \( \text{⟨subordinates⟩} \). The annotations of the form \( \text{type="xsd:string"} \) specify that the content of the \( \text{⟨firstname⟩} \) and \( \text{⟨familyname⟩} \) is simply a string. The content of the \( \text{⟨blob⟩} \) element is left unspecified while the content of the \( \text{⟨subordinates⟩} \) element is given by the type \( \text{subordinatesType} \). Here, we can see that the \( \text{⟨subordinates⟩} \) element can have arbitrary number of \( \text{⟨person⟩} \) elements. An example of a document which adheres to this type is presented on Figure 3.

\[
<\text{xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"}>
<\text{xsd:element name="person" type="personType"} />
<\text{xsd:complexType name="personType"}>
<\text{xsd:sequence}>
<\text{xsd:element name="firstname" type="xsd:string"} />
<\text{xsd:element name="familyname" type="xsd:string"} />
<\text{xsd:element name="blob"} />
<\text{xsd:element name="subordinates" type="subordinatesType"} />
</\text:xsd:sequence>
</\text:xsd:complexType>
<\text:xsd:complexType name="subordinatesType"}>
<\text{xsd:choice minOccurs="0" maxOccurs="unbounded"}>
<\text{xsd:element ref="person"} />
</\text:xsd:choice>
</\text:xsd:complexType>
</\text:xsd:schema>
\]

Figure 2: The XMLSchema specification of the documents in the application A.

An external accounting company may be interested in the structure of the company itself to conduct an audit. However, the personal information may skew the judgements of the auditors based on their personal experiences or idiosyncrasies. Therefore, one may be tempted to filter the personal information out to obtain a document in the format of the application B which is presented on Figure 4. The meaning of the specification should already be clear as it is a limited version of the specification from Figure 2. A document of this format which corresponds to the example on Figure 3 is presented on Figure 5.

The transformation between the formats of the application A and the application B can be realised by an XSLT sheet. An example of such a sheet is presented on Figure 7. This stylesheet specifies with

\[
<\text{xsl:template match="/"}>
<\text{xsl:apply-templates select="person"} />
</\text{xsl:template>
\]

that the documents which can be transformed by it must have \( \text{⟨person⟩} \) as the root element. The template starting with \( <\text{xsl:template match="person"} > \) defines how the \( \text{⟨person⟩} \) elements should be transformed. It specifies that a
<?xml version="1.0" encoding="ISO-8859-1"?>

<person>
  <firstname>John</firstname>
  <familyname>Smith</familyname>
  <blob> some information (I) </blob>
  <subordinates>
    <person>
      <firstname>George</firstname>
      <familyname>Carpenter</familyname>
      <blob> some information (II) </blob>
      <subordinates>
      </subordinates>
    </person>
    <person>
      <firstname>William</firstname>
      <familyname>Tailor</familyname>
      <blob> some information (III) </blob>
      <subordinates>
      </subordinates>
    </person>
  </subordinates>
</person>

Figure 3: An example document for the application A

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person" type="personType"/>

  <xsd:complexType name="personType">
    <xsd:sequence>
      <xsd:element name="blob"/>
      <xsd:element name="subordinates" type="subordinatesType"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="subordinatesType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="person"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>

Figure 4: The XML Schema specification of the documents in the application A.
Figure 5: An example document for the application B. This document follows the structure of the example document on Figure 3, but with the first and last names removed.

The <person> element should be transformed to the <person> element the children of which are in sequence <blob> and <subordinates>. The content of the <blob> element is just the content of the <blob> child of <person> in the input document. At the same time the content of the <subordinates> element is the result of the templates application to the <subordinates> child of the current <person> element in the input document.

Additionally we would like to make sure that subtrees rooted as <blob> are copied verbatim as these subtrees will be interpreted in further processing possibly by the application B, but maybe by an application C which is located still further in the processing chain and works for our accounting company. In this situation it may be important to ensure that the information within the <blob> element is left intact. Moreover, this may be a business requirement enforced by the accounting company—this company may need the information to conduct the actual audit. In order to express this, we can use the equality between XPath expressions on Figure 6. These expressions specify the fact that whenever

```
//blob == //blob
```

Figure 6: An equality between the XPath expressions which specifies the identity of subtrees.

ever in both documents we have <blob> nodes at the same path then the nodes are the same. This can be extended for instance to a more elaborate expressions such as

```
//blobA == //blobB
```

where we specify the content of the nodes called blobA and nodes called blobB to have the same content provided that they are at the same path in both documents. Similarly,

```
//blobA/something == //blobB/some/node
```
we can express that the content of the node something under a node blobA in one document should be the same as the content of the node node under the path blobB/some provided that the path to blobA is the same as the path to blobB.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="person"/>
  </xsl:template>
  <xsl:template match="person">
    <person>
      <blob>
        <xsl:value-of select="blob"/>
      </blob>
      <subordinates>
        <xsl:apply-templates select="subordinates"/>
      </subordinates>
    </person>
  </xsl:template>
  <xsl:template match="subordinates">
    <xsl:apply-templates select="person"/>
  </xsl:template>
</xsl:stylesheet>
```

Figure 7: An XSLT stylesheet that transforms documents of the application A to documents of the application B.

In order to adapt this data to the framework of Problem 1 stated in terms of automata one has to provide the automaton which does the required check. This can be done in four steps:

1. provide an automaton for the XML Schema on Figure 2,
2. provide an automaton for the XML Schema on Figure 4,
3. provide an automaton for the XSLT transformation on Figure 7,
4. provide an automaton for the required equality constraint on Figure 6,
5. combine the four automata.

The automata which check the XML Schema are quite obvious. The set of states is in bijection with the set of occurrences of node names in the XML Schema. Each time the children of a node are in a particular layout which is in accordance with the XML Schema specification we allow a non-deterministic transition to the state which corresponds to the specification. Additionally, the sequences are simulated as a tree which expands rightwards. The automata which follow this idea are presented on Figure 8(a) and (b).
Figure 8: (a) An automaton that corresponds to the XML Schema on Figure 2. (b) An automaton that corresponds to the XML Schema on Figure 4. (c) An (slightly simplified) automaton that corresponds to the XSLT transformation on Figure 7. (d) An automaton that checks the equality constraint on Figure 6.
The automata which follow the flow of the XSLT stylesheets can be constructed in the following way. We construct the automata using the following recipe:

- the templates of the form

```xml
<xsl:template match="element1">
  <element2>
    <element3>
      <xsl:value-of select="element4"/>
    </element3>
    <element5>
      <xsl:apply-templates select="element6"/>
    </element5>
    ...
  </element2>
</xsl:template>
```

introduce transitions of the form:

\[ \langle \text{element1}, \text{element2} \rangle (q(\text{element3}, \text{element4}), q(\text{element5}, \text{element6}), \ldots, q_\perp) \rightarrow q(\text{element1}, \text{element2}) \]

where \( q_\perp \) stands for a sequence of all the states of the form \( \langle \text{someelem}, \perp \rangle \) or \( \langle \perp, \text{someelem} \rangle \). We also introduce the rules with all the possible permutations of the states in the arguments of the operator \( \langle \text{element1}, \text{element2} \rangle \);

- all the transitions of the form:

\[ \langle \text{element1}, \perp \rangle (\ldots) \rightarrow q(\text{element1}, \perp) \]

where \( \ldots \) is any sequence of states whenever the pattern:

```xml
<xsl:template match="element1">
  <xsl:apply-templates select="element2"/>
</xsl:template>
```

occurs in the template;

- all the transitions of the form

\[ \langle \perp, \text{element1} \rangle (\ldots) \rightarrow q(\perp, \text{element1}) \]

for all the elements \text{element1} occurring in the template.

At last, the automaton to check the equalities is just the automaton which just accepts any form of the documents pair, but in case the element matches the given equality specification, the equality is checked. In particular the meaning of the equality \( //\text{blob} == //\text{blob} \) is to check all the cases when \text{blob} elements meet.
5 The emptiness problem

The emptiness problem for the automata considered here is undecidable [Tre00]. This, however, is a result of the fact that too many state changes are allowed under an equality check. We conjecture that the following class of the automata has the emptiness problem decidable.

**Definition 11** (automata with fixed number of changes)
Let us fix $k > 0$. We say that an automaton $A$ has number of changes less than $k$ if for each $A$-run $R$ and each $\gamma$ such that $\text{constr}_R(\gamma)$ is non-empty the number of paths $\gamma' \succ \gamma$ such that $\{q_1, \ldots, q_l\} \neq \{q\}$ where

$$\text{label}_R(\gamma') = f(q_1, \ldots, q_l) \mapsto q$$

is less than $k$.

This class of automata is in fact very limited as virtually all the state changes can be realised either above a constraint or in a very close vicinity of a constraint transition. Therefore, we also conjecture that the emptiness problem for another, less constrained class of automata is decidable as well.

**Definition 12** (relaxed automata with fixed number of changes)
Let us fix $k > 0$ and a state $q_{\text{blank}}$. We say that an automaton $A$ has number of changes less than $k$ if for each $A$-run $R$ and each $\gamma$ such that $\text{constr}_R(\gamma)$ is non-empty the number of $\gamma' \succ \gamma$ such that $\{q_1, \ldots, q_l\} \neq \{q, q_{\text{blank}}\}$ where

$$\text{label}_R(\gamma') = f(q_1, \ldots, q_l) \mapsto q$$

is less than $k$.

These restrictions indeed limit the number of state changes. Another interesting subcase is when we allow only comparisons between brothers. In particular the example equality on Figure 6 transforms to this kind of check as can be seen on Figure 8(d). We have a theorem here:

**Theorem 5.1** (T)
The emptiness problem for the class of automata in which

- we restrict the constraints so that their paths have only length 1,
- we loosen the constraint for $T$ in Definition 4 and allow this to be any graph,

is decidable.

**Proof:**
The decidability in this case can be obtained using the known techniques for automata with constraints, but with no coordinates see [CDG+97].

6 Conclusions

In this paper we presented a class of automata with equality constraints on coordinates. We showed that this class of automata is useful in checking the properties of XSLT stylesheets. Additionally, we presented the current state of the art concerning the decidability of the emptiness problem for this class of automata.
References


