A Generic XML-Generating Metacompiler

Hassan Aït-Kaci

hak@acm.org

February 16, 2014

Abstract

This describes a feature of Jacc,[1] a Java-based system in the fashion of Yacc,[2] the well-known metacompiler, where a minimalistic set of simple annotations may be specified on a few grammar's rules and terminal symbols to guide the automatic generation of code in XML format. The annotations basically specify how to produce an XML construct out of the bits and pieces of a concrete syntax tree (CST). The information is then used at parse-time by the generated parser to build the actual XML tree in accordance with the specified patterns. This annotation-driven process is one of *tree transduction* (from CST to XML tree). Compilers generated from such annotated Jacc grammars can generate XML code for a wide class of possible formats depending on the syle of annotation chosen for serializing a piece of syntax corresponding to a few grammar rules and terminals. In this document, we present the simple notation Jacc uses for such annotations decorating a Yacc-style BNF grammar, and its operational semantics. We illustrate each construct with examples of its use. We also discuss benefits and limitations of the current system.

KEYWORDS: Metacompilers, XML, Annotation-driven XML pattern generation

Contents

[1] Just another compiler compiler.

[2] Yet another compiler compiler.

1 Introduction

1.1 Motivation

One of the nice consequences of the advent of the W3C eXtended Markup Language (XML) is that is has become the universal format for producing code.

1.2 Approach

Our approach is simple: we use a simple grammar annotation scheme that drives XML-code generation.

1.3 Organization of contents

The rest of this document is organized as follows. In Section 2, we review extant work in relation to the work presented here. Section 3 overviews the functionality of `Jacc`. In Section 4, the XML annotation notation is described in detail. Section **??** illustrates how to use `Jacc` using a complete annotated grammar example. We conclude in Section 6 with some comments on the usefulness and future potential evolution of this tool.

2 Review of related work

3 Quick overview of `Jacc`

At first sight, `Jacc` may be seen as a "100% Pure Java" implementation of an LALR(1) parser generator [1] in the fashion of the well known UNIX tool `Yacc`—"yet another compiler compiler" [4]. However, `Jacc` is much more than... *just another compiler compiler*: it extends `Yacc` to enable the generation of flexible and efficient Java-based parsers and provides enhanced functionality rarely available in other similar systems.

The fact that `Jacc` uses `Yacc`'s metasyntax makes it readily usable on most `Yacc` grammars. Other Java-based parser generators all depart from `Yacc`'s format, requiring nontrivial metasyntactic preprocessing to be used on existing `Yacc` grammars—which abound in the world, `Yacc` being by far the most popular tool for parser generation. Importantly, `Jacc` is programmed in pure Java—this makes it fully portable to all existing platforms, and immediately exploitable for web-based software applications.

`Jacc` further stands out among other known parser generators, whether Java-based or not, thanks to several additional features. The most notable are:

- `Jacc` uses the most efficient algorithm known to date for its most critical computation (*viz.*, the propagation of LALR(1) lookahead sets). Traditional `Yacc` implementations use the method originally developed by DeRemer and Penello [3]. `Jacc` uses an improved method due to Park, Choe, and Chang [5], which drastically ameliorates the method of by DeRemer and Penello. To this author's best knowledge, no other (available) Java-based metacompiler system implements the Park, Choe, and Chang method [2].

- `Jacc` allows the user to define a complete class hierarchy of parse node classes (the objects pushed on the parse stack and that make up the parse tree: nonterminal and terminal symbols), along with any Java attributes to be used in semantic actions annotating grammar rules. All these attributes are accessible directly on any pseudo-variable associated with a grammar rule constituents (*i.e.*, `$$`, `$1`, `$2`, *etc.*).

- `Jacc` makes use of all the well-known conveniences defining precedences and associativity associated to some terminal symbols for resolving parser conflicts that may arise. While such conflicts may in theory be eliminated for any LALR(1) grammar, such a grammar is rarely completely obtainable. In that case, `Yacc` technology falls short of providing a safe parser for non-LALR grammar. Yet, `Jacc` can accommodate any such eventual unresolved conflict using non-deterministic parse actions that may be tried and undone.

- Further still, `Jacc` can also tolerate non-deterministic tokens. In other words, the same token may be categorized as several distinct lexical units to be tried in turn. This allows, for example, parsing languages that use no reserved keywords (or more precisely, whose keywords may also be tokenized as identifiers, for instance).

- Better yet, `Jacc` allows dynamically (re-)definable operators in the style of the Prolog language (*i.e.*, at parse-time). This offers great flexibility for on-the-fly syntax customization, as well as a much greater recognition power, even where operator symbols may be overloaded (*i.e.*, specified to have several precedences and/or associativity for different arities).

- `Jacc` supports partial parsing. In other words, in a grammar, one may indicate any nonterminal as a parse root. Then, constructs from the corresponding sublanguage may be parsed independently from a reader stream or a string.

- `Jacc` automatically generates a full HTML documentation of a grammar as a set of interlinked files from annotated `/**...*/ javadoc`-style comments in the grammar file, including a navigatable pure grammar in "`Yacc` form," obtained after removing all semantic and serialization annotations, leaving only the bare syntactic rules.

- `Jacc` may be directed to build a parse-tree automatically (for the concrete syntax, but also for a more implicit form which rids a concrete syntax tree of most of its useless information). By contrast, regular `Yacc` necessitates that a programmer add explicit semantic actions for this purpose.

- `Jacc` supports a simple annotational scheme for automatic XML serialization of complex Abstract Syntax Trees (ASTs). Grammar rules and non-punctuation terminal symbols (*i.e.*, any meaning-carrying tokens such as, *e.g.*, identifiers, numbers, *etc.*) may be annotated with simple XML templates expressing their XML forms. `Jacc` may then use these templates to transform the Concrete Parse Tree (CST) into an AST of radically different structure, constructed as a `JDOM` XML document.[3] This yields a convenient declarative specification of a tree transduction process guided by just a few simple annotations, where `Jacc`'s "sensible" behavior on unannotated rules and terminals works "as expected." This greatly eases the task of retargeting the serialization of a language depending on variable or evolving XML vocabularies.

[3] http://www.jdom.org/

With `Jacc`, a grammar can be specified using the usual familiar `Yacc` syntax with semantic actions specified as Java code. The format of the grammar file is essentially the same as that required by `Yacc`, with some minor differences, and a few additional powerful features. Not using the additional features makes it essentially similar to the `Yacc` format.

For details on how `Jacc` extends `Yacc` to support Prolog-style dynamic operators, see Section **??**. For instructions on how to organize a `Jacc` grammar, please refer to the documentation of the grammar format, or to the description of grammar commands.[4] If you wish to use `Jacc`, follow these simple steps. You may also want to peruse the code of `Jacc` grammar examples. References For detailed explanations of most constructions and algorithms used by this package, please refer to the following.

4 XML serialization annotation

We need a means to annotate a `Jacc` grammar so as to ease and automate the process of specifying an XML serialization for the language defined by the grammar. The way we proceed is by annotating some rules and terminals to produce an XML form built out of those XML forms built for the constituents of the CST (*i.e.*, from a terminal's contents or a rule's RHS).

To this end, `Jacc` will come handy. This section describes a (meta-)grammar for a simple annotation language meant to enable passing XML formatting information from a `Jacc` grammar to a `Jacc` parser. This language is that of the forms that go between square brackets either in the **`%xmlinfo`** command annotating a terminal or appearing in a rule being annotated for XML conversion for serialization purposes. Doing this gives us great flexibility for extending or modifying the annotation meta-syntax simply by:

1. modifying the `Jacc` grammar source file;
2. running the `jacc` command on it to regenerate the `XmlAnnotationParser` Java source;
3. recompiling.

Et voilà ! ...

4.1 Basic annotation notation

We first introduce the basic annotation notation for the very common case when the XML tree to be constructed from the CST is homomorphic to the CST in that it only needs information that is local to the CST node. We will extend this notation later when the tree construction is heteromorphic, needing information from below this node.

In order for the parser of the annotation notation to stay small and light-weight, as well as avoiding ambiguity and stay strictly within LALR(1) recognition power, we will adopt the following very simple keyword-driven syntax. For example:

[4] http://hassan-ait-kaci.net/hlt/doc/hlt/jaccdoc/000_START_HERE.html

```
A0  :  A1 A2 A3 A4
    [
       nsprefix    :  "foo"
       localname   :  "Azero"
       attributes  :  {a = "bar blah", b="blech"}
       children    :  (2, 3)
    ]
    ;
```

means that the XML form of an $\mathtt{A}_0$ node (created when the parser uses this grammar rule bottom-up) will look like:

```
<foo:Azero a="bar blah" b="blech">
(XML form of A2)
(XML form of A3)
</foo:Azero>
```

In such an annotation, appearing in any order, the notation `"keyword : value"` is such that keyword is an admissible keyword. An admissible keyword is such that there may be at most one occurrence per annotation of **`nsprefix`**, **`localname`**, **`attributes`**, or **`children`**.

Such an admissible keyword is followed by a value, which may be either an identifier, a single- or double-quoted string, or a list between curly braces {...}, or parentheses `(...)`, the nature of this list's brackets and elements depending on the keyword (see the annotation grammar for details).

The annotation is meant to be light-weight. So, all these keywords may be abbreviated to any non-empty case-insensitive prefix of their full form, and some punctuation may be used interchangeably or simply omitted: *e.g.*, the '`:`' separating keywords and values, the '`,`' separating list elements, as well as unnecessary quotes, are all in fact optional. The following key/value separator symbols may be used: '`:`' '=' '->' '=>' or they may be simply omitted. Similarly, the following list separator symbols may be used: '`,`' (comma), '`;`' (semicolon), or they may be simply omitted.

For example, the annotation shown above could as well be written as follows:

```
A0  :  A1 A2 A3 A4
    [
       NS  :  foo
       LO  :  Azero
       AT  :  {a -> 'bar blah'; b -> blech}
       CH  :  (2 3)
    ]
    ;
```

4.2 More complex annotation notation

The simple notation above is all one needs in many common cases: it works whenever the XML serialization pattern is constructible only from the immediate constituents of the rule's LHS (`A0`)—*i.e.*, the XML trees of the rule's RHS symbols (when $n > 0$). It is, however, insufficient for expressing XML serialization patterns that depend on sub-elements contained within those of the

XML serialization of the RHS symbols. The simple case is called *homomorphic* tree transduction, while the more complex case is called *heteromorphic* tree transduction.[5]

A more elaborate XML annotation is described next. It extends the above basic notation by allowing the values of attributes and children in the annotation to take on more complex forms. These forms denote references to existing XML constructs within the XML trees already built for the CST children of this node. We will illustrate the meaning of these extended forms using simple color-coded examples. These show how the basic notation for homomorphic tree-transduction annotations for attribute and children can be extended to express some heteromorphic tree-transduction as well.

4.2.1 Children annotation

The full form of the annotation expression for specifying children is:

CH:(... $\mathtt{w}_1$ **...** $\mathtt{w}_n$**c[**$\mathtt{x}_1$**. ... .**$\mathtt{x}_m$**]/a ...)**

of which only **c** is mandatory. The four parts of a child specification expression are such as described next.

1. The wrapper path $\mathtt{w}_1 \ldots \mathtt{w}_n$ is optional: each wrapper $\mathtt{w}_i$ is a pair made of a (unquoted, single-quoted, or double-quoted) string (an XML tag), followed by a distribution marker, which is either a dot (‘`.`’), or an asterisk (‘`*`’). Using a dot triggers single wrapping, while using an asterisk triggers distributive wrapping (at the nesting level specified).

2. The child—there are two cases:

   (a) in a rule’s annotation, **c** is a positive integer and denotes a position in the rule’s RHS (*i.e.*, a position in the CST) and refers to the XML tree corresponding to the child CST at this position;

   (b) if not a number, **c** must be a *special form*. In this case, there may be nothing trailing after **c**; *i.e.*, **[**$\mathtt{x}_1$**. ... .**$\mathtt{x}_m$**]/a** is empty.

3. The XML tree path **[**$\mathtt{x}_1$**. ... .**$\mathtt{x}_m$**]** is optional; if not empty, it denotes a path in the XML tree. This path corresponds either to a single XML component of a CST node, or the set of all XML components of a CST node. In the first case (single component), each $\mathtt{x}_j$ is a positive integer denoting a component position in the XML tree rooted in this referring CST child. This is also true in the second case (all children), except for the last ($\mathtt{x}_m$) which is then equal to **0**. In particular, it may be reduced to just the form **c[0]**, in which case it refers to the set of all the XML components of the XML form of the CST node in position **c**.

4. The attribute reference **/a** is optional; when present, **a** is a (possibly unquoted, single-quoted, or double-quoted) string; it must be the name of an XML attribute in the ultimate XML tree referred to by **c[**$\mathtt{x}_1$**. ... .**$\mathtt{x}_m$**]**, and denotes the string content making up the value of that XML attribute. When an attribute reference is specified, the XML tree path may not end with **0** (since it then does not refer to one node, but to a set of nodes).

[5] The Greek etymology of the words says precisely that: *“homo-morphic” = “similar form”* (from the Greek “$\eta o\mu o$-$\mu o\varrho\varphi o\varsigma$,” meaning “same shape”), and *“hetero-morphic” = “of dissimilar form”* (from the Greek “$\eta\varepsilon\tau\varepsilon\varrho o$-$\mu o\varrho\varphi o\varsigma$,” meaning “different shape”).

4.2.2 Examples of children annotation

Basic children annotation The notation:

```
CH:(2, 4)
```

specifies that the XML children are, in this order:

1. the XML form of `2`nd child CST,
2. the XML form of `4`th child CST.

Extended children annotation

- *Grandchild reference*—the notation:

  ```
  CH:(2[1], 4[2])
  ```

  specifies that the XML children are, in this order:

  1. the `1`st XML component of the XML form of `2`nd child CST,
  2. the `2`nd XML component of the XML form of `4`th child CST.

- *Grandchildren set reference*—the notation:

  ```
  CH:(2[0], 4[2])
  ```

  specifies that the XML children are, in this order:

  1. all the XML components of the XML form of `2`nd child CST,
  2. the `2`nd XML component of the XML form of `4`th child CST.

- *Descendant reference*—the notation:

  ```
  CH:(2[1.4], 1[2.1.3])
  ```

  specifies that the XML children are, in this order:

  1. the `4`th XML component of `1`st XML component of the XML form of `2`nd child CST,
  2. the `3`rd XML component of `1`st XML component of `2`nd XML component of the XML form of `1`st child CST.

- *Attribute reference*—the notation:

  ```
  CH:(2[1.4]/foo)
  ```

  specifies that the only XML child is the string value of the attribute named `foo` of the `4`th XML component of `1`st XML component of the XML form of `2`nd child CST.

- *Wrappers*—the notation:

  ```
  CH:(foo.2, bar.fuz.4)
  ```

specifies that the XML children are, in this order:

1. <**foo**>*(XML form of 2nd child CST)*</**foo**>
2. <**bar**><**fuz**>*(XML form of **4**th child CST)*</**fuz**></**bar**>

By default, wrappers do not distribute over their contents. In other words, the resulting form will be one with all the contents wrapped in a single nesting of wrappers. If it is desired to override this default behavior and actually distribute a wrapper tag path over the sequence making up the contents being wrapped, then one uses an asterisk ('`*`') instead of a dot ('`.`'), as in, *e.g.*:

```
CH:(foo*2, bar*fuz.4)
```

Thus, using an asterisk rather than a dot in specifying a wrapper path triggers one of three things depending on whether the contents being wrapped is:

1. *nothing*—in which case nothing is generated;
2. *a single XML element*—in which case the wrapped single element is generated;
3. *a sequence of XML elements*—in which case the corresponding sequence of wrapped elements is generated.

4.2.3 Attributes annotation

The full form of the annotation expression for specifying an attribute is:

AT:{... **foo=c**[$\mathbf{x}_1$. ... .$\mathbf{x}_m$]**/a** ...}

of which only **c** is mandatory.

- If [$\mathbf{x}_1$. ... .$\mathbf{x}_m$]**/a** is missing, then **c** may be only one of:
  1. a literal string—*e.g.*, **"bar"**; or,
  2. a special form—*i.e.*, **$VALUE** or **$TEXT**.
- If [$\mathbf{x}_1$. ... .$\mathbf{x}_m$] is present, the $\mathbf{x}_i$'s are a sequence of dot-separated positive integers, an XML tree path referencing an XML subtree. Then, the annotation must be that of a rule and **c** must be a positive integer denoting the position a child CST for the current rule (a position in the rule's RHS). It refers to the XML tree of child CST at that position.

  If **/a** is present, it must be the name of an attribute in the XML tree so referenced. This annotation denotes the string value of this attribute in that XML tree. If **/a** is missing, then the annotation denotes the text content of the XML tree so referenced.

4.2.4 Examples of attribute notation

Basic attribute annotation The notation:

AT:{**foo="bar"**}

sets the attribute named **foo** to the literal string value **"bar"**.

Extended attribute annotation

- *Child's text value*—the notation:

  AT:{**foo=3**}

  sets the attribute named **foo** to the text value of the XML form of **3**rd child CST.

- *Child's attribute value*—the notation:

  AT:{**foo=3/bar**}

  sets the attribute named **foo** to the value of the attribute named **bar** in the XML form of **3**rd child CST.

- *Descendant's text value*—the notation:

  AT:{**foo=3[1.2]**}

  sets the attribute named **foo** to the text value of the **2**nd XML component of the **1**st XML component of the XML form of the **3**rd child CST.

- *Descendant's attribute value*—the notation:

  AT:{**foo=3[1.2]/bar**}

  sets the attribute named **foo** to the value of the attribute named **bar** located in the **2**nd XML component of the **1**st XML component of the XML form of the **3**rd child CST.

- *Terminal value*—in a terminal's annotation only, the notation:

  AT:{**foo=$VALUE**}

  sets the attribute named **foo** to the value of the terminal node actually parsed.

4.2.5 Interpreted special forms

In addition to the above notation (and default behavior), we provide the following conveniences to specify finer details on the XML appearance from the information present in the CST thanks to the following built-in special forms, which all starting with a dollar sign ('`$`'), followed by the (case-insensitive) form identifier and possible arguments between parentheses and separated by a legal list separator; namely, blank space, '**,**' (comma), or '**;**' (semicolon).

Extracting the value of a terminal The notation **$VALUE** may appear in an XML annotation expression for either a rule or a terminal whenever the CST construct it refers to is that of a terminal. For example:

- *an attribute value string; e.g.*, the notation:

  ```
  %xmlinfo SHTOONG [ L:"BAR" N:"Foo" A:{ fuz = $VALUE } ]
  ```

  specifies that a terminal symbol **SHTOONG** with print value **"Gloop"** will be serialized as follows:

```
<Foo:BAR fuz="Gloop"/>
```

- *a single XML content string; e.g.*, the notation:

  ```
  %xmlinfo SHTOONG [ L:"BAR" N:"Foo" C:( $VALUE ) ]
  ```

  specifies that a terminal symbol `SHTOONG` with print value `"Gloop"` will be serialized as follows:

  ```
  <Foo:BAR>Gloop</Foo:BAR>
  ```

Concatenating pieces of text Wherever text is expected, we may use the notation `$TEXT(...)` to denote the text string resulting from the concatenation of the text strings denoted by its arguments, each of which may be either a literal (possibly single- or double-quoted) string, or a reference to a text value deeper in a descendant CST's XML structure using the XML tree reference notation $\mathtt{c[x_1...x_n]/a}$, where the $\mathtt{[x_1...x_n]}$ and `/a` parts are optional.

This construct comes handy for composing a text string on the fly to make up the text value of a child or an attribute. For example, given the annotations in Figure 1, the piece of `Entry` syntax:

```
%xmlinfo  ID
          [
               L  :  "Identifier"
               A  :  { name = $VALUE}
          ]

%xmlinfo  STR
          [
               L  :  "String"
               A  :  { value = $VALUE}
          ]

Type      :    ID ':' ID
          [
               L  :  "Type"
               A  :  { general = 1/name special = 3/name }
          ]
          ;

Entry     :    STR '@' Type
          [
               L  :  "Place"
               A  :  { type = $TEXT( "[" 3/special "]" 3/general ) }
               C  :  ( 1/value )
          ]
          ;
```

Figure 1: Example of annotations using the `$VALUE` and `$TEXT` special forms

```
bar@less:top
```

gets serialized as:

```
<Place type="[top]less">bar</Place>
```

4.3 Checking annotation consistency

We need to enforce consistent number referencing in the tree addresses used in the notation—*i.e.*, the numbers that refer to RHS nodes and XML elements (the $\mathtt{c}_i$'s and the $\mathtt{x}_i$'s below). Indeed, they should (be made to) obey the following necessary conditions (all easy to justify):

- **Condition 1:** An annotation for a terminal, or for a rule with an empty RHS, should not be allowed to use a tree address in any attribute specifier (only symbol, quoted string, or number). A terminal's annotation **CH** may only contain wrappers and a reference to **$VALUE**.
- **Condition 2:** In an annotation **AT:{...}**, the name of an attribute following an XML tree reference must be a legal attribute of the element so referenced.
- **Condition 3:** In a rule annotation `CH:(...`$\mathbf{c}_i$`[...]...)`, the number of $\mathbf{c}_i$'s must be between 1 and the length of the rule's RHS.
- **Condition 4:** In an annotation `CH:(...)`, two distinct occurrences of XML content references must not be allowed to be one another's prefix or duplicate address. In other words, no tree address may occur more that once in the same annotation; and, whenever a tree address occurs in an annotation, none of its prefixes may occur in the same annotation.

  In other words, whenever the child path expression $\mathbf{c[x}_1\mathbf{.\ ...\ .x}_n\mathbf{]}$ occurs in a `CH` annotation, then neither $\mathbf{c[x}_1\mathbf{.\ ...\ .x}_{n-1}\mathbf{]}$, nor $\mathbf{c[x}_1\mathbf{.\ ...\ .x}_{n-2}\mathbf{]}$, ..., nor $\mathbf{c[x}_1\mathbf{]}$, nor **c** may be allowed to occur in the same `CH` annotation.

  For example, both the `CH` annotations `CH:`**(1 2)** and `CH:`**(1 2[1] 2[2])** are legal; however, neither `CH:`**(1 2 1)** nor `CH:`**(1 2[1] 1[2])** are.
- **Condition 5:** Whenever a tree address of the form $\mathbf{c[x}_1\mathbf{.x}_2\mathbf{.\ ...\ .x}_n\mathbf{]}$ occurs, then for it to be consistent, this entails that the XML form of the CST node referenced by **c** must consist of exactly one XML element—as opposed to none or many. This is true iff the referenced RHS symbol is either a value-carrying terminal, or a non-terminal all of whose possible XML forms are each single XML elements. This must be verified statically at grammar analysis time.
- **Condition 6:** Whenever a tree address of the form $\mathbf{c[x}_1\mathbf{.x}_2\mathbf{.\ ...\ .x}_n\mathbf{.0]}$ occurs, (with $n \geq 0$), it refers to the set of all children of the node $\mathbf{c[x}_1\mathbf{.x}_2\mathbf{.\ ...\ .x}_n\mathbf{]}$. Therefore, it may not be followed with an attribute, and no other tree address other than this one may be of the form $\mathbf{c[x}_1\mathbf{.x}_2\mathbf{.\ ...\ .x}_n\mathbf{....]}$. This goes also for $n = 0$; for example, `CH:`**(2[1] 2[0])** is forbidden since the 1st XML child of the 2nd CST child is referred twice (first explicitly individually, then implicitly in the set of all children of the same node).

Violation of any of these conditions at parser-generation time should raise an exception and be reported as an error. If all these conditions hold, then the code for the method `xmlify(Element container)` defined in the class `ParseNode`, and the method `createXmlForm(ParseNode node, Element root)` defined in the class `XmlInfo`, is guaranteed to work safely.

5 Example

6 Conclusion

6.1 Recapitulation

6.2 Further work

References

[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] CHOE, K.-M. Private communication (`choe@compiler.kaist.ac.kr`). Korean Advanced Institute of Science and Technology, Seoul, South Korea, December 2000.

[3] DEREMER, F., AND PENELLO, T. Efficient computation of lookahead sets. *ACM Transactions of Programming Languages and Systems 4*, 4 (October 1982), 615–749.

[4] JOHNSON, S. C. Yacc: Yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Labs, Murray Hill, NJ (USA), 1975. Reprinted in the *4.3BSD Unix Programmer's Manual, Supplementary Documents 1, PS1:15. UC Berkeley, 1986.*

[5] PARK, J., CHOE, K.-M., AND CHANG, C. A new analysis of LALR formalisms. *ACM Transactions of Programming Languages and Systems 7*, 1 (January 1985), 159–175.