Automated verification of termination certificates

Kim Quyen LY

University Joseph Fourier

2 March 2012
Supervisor: Frédéric BLANQUI

Outline

Goal

develop a safe, efficient and modular termination certificate verifier

our solution:

- CPF: new common format for termination certification
- write a verifier in Coq
- prove its correctness using the CoLoR library
- extract it to OCaml

CPF

the CPF format is regularly modified and extended with new features, it is useful to have a tool that can automatically generate in OCaml and Coq:

- data structures
- parsers
- pretty-printers

for that format

What did I do until 17 November 2011?

- generator of Coq type definitions from XSD
- generator of OCaml type definitions and parsing functions from XSD
- replaced modules by records in some CoLoR files
- translation of CPF types into CoLoR types
- definition of a certificate verifier for polynomial interpretations
- correctness proof almost finished for polynomial interpretations

Work plan for November 2011 - October 2012

- November 2011: finish the new implementation of the OCaml and Coq type definitions generator from XSD (almost done)
- December 2011: finish the correctness proof for polynomial interpretations on integers (not done)
- January 2012: extraction to OCaml, linking with CPF parser and testing on TPDB (not done)
- February 2012 - October 2012: extension to other termination techniques

Outline

XML

termination certificates are given as XML files

```
<root>
  <child1 attribute1="value1">
    <subchild> .... </subchild>
  </child1>
  <child2>
    <subchild> .... </subchild>
  </child2>
</root>
```

abstractly, an XML file is a tree whose nodes are tagged and may have attributes (leaves are strings)

XML Schema (XSD)

an XSD file describes a class of XML files by defining the possible tags and attributes, and how tagged elements can be composed

XSD type = set of XML elements

It is itself defined as an XML file! with the following tags:

- `<sequence>XSD_type1 XSD_type2 ...</sequence>`: product type
- `<choice>XSD_type1 XSD_type2 ...</choice>`: union type
- `<group name ="<name>">XSD_type</group>`: names a type
- `<element name="<tag>">XSD_type</element>`: declares a tag, its attributes and its possible sons

remark: XSD definitions need not be ordered and can be forward or backward referenced

XSD example (part 1/2)

```
<xs:group name="symbol">
  <xs:annotation>
    <xs:documentation>is used as a function symbol in terms, orderings, ....</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element ref="name"/>
    <xs:element name="sharp">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="symbol"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="labeledSymbol">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="symbol"/>
          <xs:group ref="label"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
```

Representation in Coq?

`choice` elements are represented by inductive types
`sequence` elements are represented by lists

- one big inductive type?
  ⇒ complex induction principle and proofs

- several small inductive types
  ⇒ XSD definitions need to be ordered

Outline

Dependency relation

type expressions: $T = C | T \Rightarrow T$
type definition for a type constant C: a type for each constructor

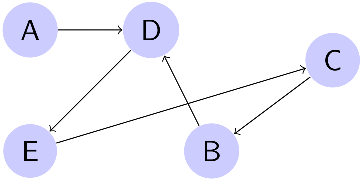
C *def-depends* on D, written $C \leadsto D$, if:
C has a constructor in the type of which D occurs

let $\geq$ ($\simeq$) be the transitive (reflexive and symmetric) closure of $\leadsto$

- two types C and D *depend* on each other if $C \simeq D$
- a type C can be defined *before* D if $C < D$

Representing finite relations as boolean matrices

To present the dependency relation $C \rightsquigarrow D$ we decided to used the adjacency matrix



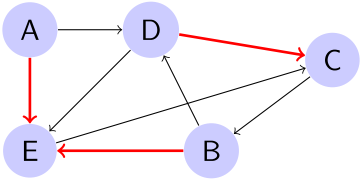
$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

1 if the edge is there

0 if there is no edge

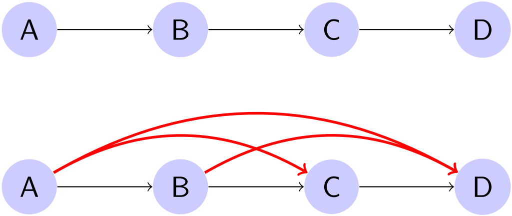
Computation of the transitive closure

for example: a directed graph G, if we have a path from $A \rightarrow D$ and $D \rightarrow E$ then we will have a path from $A \rightarrow E$.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Computation of the transitive closure



define: $R^0 = R$; for $i > 0$,
$R^i = R^{i-1} \cup \{(a, c) \mid \exists b \text{ where } (a, b) \in R^{i-1} \text{ and } (b, c) \in R^{i-1}\}$

$\mathrm{TC}(R) = \cup_{i \in \mathbb{N}} R^i$

$\mathrm{TC}(R)$ is a transitive closure of R if $(a, b) \in R^j$ and $(b, c) \in R^k$, then from composition's associativity $(a, c) \in R^{j+k}$

Computation of equivalence classes

An equivalence class is a set of elements that are all equivalent.

if we have a path $A \rightarrow B$ and also $B \rightarrow A$.

An equivalence relation statisfying three properties: reflexivity, symmetry and transitivity.

Boolean matrices:
same class $A \leftrightarrow B \Rightarrow (true, true)$
there is a path from $A \rightarrow B \Rightarrow (true, false)$
there is a path from $B \rightarrow A \Rightarrow (false, true)$
A and B are not comparable $\Rightarrow (false, false)$

Ordering of equivalence classes

Problem: the dependency relation is not a total order: some elements may be incomparable

⇒ we need to compute a linear/total extension of that partial order

Computation of a linear extension

Compute the linear in boolean matrix by searching between two nodes a and b. If there is no edge between them then we add an edge $a \to b$ and compute an transitive closure

Other problems in Coq

some type definitions need to be inlined to be accepted by Coq

```
Definition vector := list coefficient.
Inductive coefficient :=
| Coefficient_vector : vector -> coefficient.
```

is not valid in Coq
instead we use:

```
Definition vector := list coefficient.
Inductive coefficient :=
| Coefficient_vector : list coefficient -> coefficient.
```

Other problems in Coq

types taking lists as arguments are not seen as recursive by Coq

$\Rightarrow$ we need to prove by hand the induction principle for these types

(can this be automated? yes, see works on inductive types)

Outline

What did I do?

- finished the new implementation of the Coq type definitions generator from XSD

Work plan

- March 2011: finish the correctness proof for polynomial interpretations on integers. Extraction to OCaml, linking with CPF parser and testing on the Termination Problem Data Base (TPDB)
- April 2012 - October 2012: extension to other termination techniques

Thank you for your attention!!!