Flow graph parsing and its application in (business) process management and SOA

Hagen Völzer
Joint work with Jussi Vanhatalo and Artem Polyvyanyy
parse | pärs |
verb [ trans. ]
analyze (a sentence) into its parts and describe their syntactic roles.
  • Computing analyze (a string or text) into logical syntactic components, typically in order to test conformability to a logical grammar.
  • examine or analyze minutely: he has always been quick to parse his own problems in public.

noun Computing
an act of or the result obtained by parsing a string or a text.

ORIGIN mid 16th cent.: perhaps from Middle English pars [parts of speech,] from Old French pars ‘parts’ (influenced by Latin pars ‘part’).
Flow Graph Parsing

- Unique source, unique sink
- Every node is on a path from the source to the sink

Flow graph

- Decomposition into hierarchy of Single-Entry-Single-Exit (SESE)-fragments
- A fragment has the same properties as a flow graph
Outline

- What?
  - Flow graph parsing

- Why?
  - Use cases for flow graph parsing

- How?
  - Requirements
  - Approach
  - Algorithms
Workflow graphs

- Nodes are either *gateways* (x₁, p₁, …) or *tasks* (a₁, a₂, …)
- Control state is represented by *tokens* on the edges
Use Case 1: BPMN to BPEL Translation

Business Process Modeling Notation (BPMN)

Parse tree

Business Process Execution Language (BPEL)
Control flow errors

- **Deadlock (A token is stuck)**
- **Lack of synchronization (Two tokens or more)**

![Diagram of control flow errors with nodes and edges representing fork, join (synchronization), decision, and merge (no synchronization).]
Use Case 2: Detection of Control-Flow Errors

- **Divide-and-conquer technique**
  - Speeds up a base technique
    - E.g. a state-space exploration

1. **Decomposes** a workflow graph into its fragments
2. **Detects** errors for each fragment in isolation (Absence of errors here is compositional w.r.t. fragments.)

- May **reduce** the number of states significantly
- **Fast heuristics** are known for many fragments occurring in practice
Use Case 3: Subprocess Detection
Subprocess Detection
Use Case 4: Compare & Merge

- Compound changes rather than elementary changes
  - less changes, abstraction of edge changes, categories of changes

- Visualization according to the fragment structure of the process model
  - easier to handle for the business user

- Resolution of changes
  - Easy to integrate with control-flow analysis
  - Assure that a change does not introduce errors
Other Use Cases for Parsing

- Process Model Refactoring
- Process Similarity Search
- Layout, printing large graphs
- Understanding large process models
- Data-flow analysis
- Runtime Optimization

- Other models: Control flow graphs, Petri nets, Circuits
Outline

- **What?**
  - Flow graph parsing

- **Why?**
  - Use cases for flow graph parsing

- **How?**
  - Requirements for parsing
  - Approach
  - Algorithms
Requirement 1: Uniqueness

- Naïve parsing approach:
  - Find a fragment and parse the fragment until there is no more unparsed fragment
  - Finding is non-deterministic, parse tree is not unique

- The parse tree should be unique
  - The same BPMN model is always translated to the same BPEL model
  - Important for structurally comparing two models
Requirement 2: Modularity

- **Motivation:** A local change in BPMN translates into a local change in BPEL
  - Again important for comparing two models

- **Modular:**
  - Replacing a fragment with another fragment changes only the respective subtree in the parse tree

- Naïve parsing is not modular
Canonical Fragments

Parse tree is a hierarchy of fragments in which any two fragments do not overlap. Some fragments must be excluded from a parse tree.

Exclude fragments that overlap with some other fragment.
  - Such a fragment is called non-canonical.
  - The non-maximal sequences are the non-canonical fragments here.
The Normal Process Structure Tree (NPST)

- The Tree of canonical fragments
- The NPST is unique and modular
- Always search for sequences last, do not consider non-maximal sequences
- Extends work by Johnson et al. (1994)
  - Can be computed in linear time (using DFS techniques)
**Requirement 3: Granularity (Finer decomposition)**

- **Motivation:** Translate more BPMN models into BPEL
  - Also helpful in control-flow analysis and other use cases

- **Our contribution is the** **Refined Process Structure Tree**
  - Extends work by Tarjan and Valdes (1980)
  - More fine-grained than any previous technique
  - Proved uniqueness and modularity
  - Can be computed in linear time
Outline

- **What?**
  - Flow graph parsing

- **Why?**
  - Use cases for flow graph parsing

- **How?**
  - Our contribution: The Refined Process Structure Tree
  - Approach to computation: Using the triconnected components
  - An even more elegant algorithm
  - Lifting some assumptions
    - Dealing with separation points
    - Dealing with multiple sources/sinks
Relaxed Notion of a Fragment

The commonly used notion:

- A *fragment* is a connected subgraph that has
  - exactly one entry edge, and
  - exactly one exit edge.

Relaxed notion:

- A *fragment* is a connected subgraph that has
  - exactly one entry node, and
  - exactly one exit node.
More Precisely [Tarjan and Valdes, 1980]:

- A **fragment** $F$ is a connected subgraph that has
  - exactly two boundary nodes,
  - one entry, and one exit

- A boundary node is an **entry** of $F$ if
  - all incoming edges are outside $F$, or
  - all outgoing edges are inside $F$

- A boundary node is an **exit** of $F$ if
  - all incoming edges are inside $F$, or
  - all outgoing edges are outside $F$

- (Entry cannot be used to get out, exit cannot be used to get in)

- Each edge is a **trivial** fragment

*Not a fragment!*

*These boundary nodes are neither entries nor exits*
Non-Canonical and Canonical Fragments

- **Non-canonical** fragments overlap with some fragment

  - Non-maximal sequences
  - Non-canonical bond fragments

- **Canonical** fragments do not overlap and thus they form a hierarchy

  - Maximal sequence
  - Canonical bond fragments
The Refined Process Structure Tree (RPST)

- As the canonical fragments do not overlap, they form a hierarchy.
- The **Refined Process Structure Tree** is the tree of canonical fragments.
Granularity

- The RPST is:
  - More fine-grained than
    - the NPST
    - the parse tree by Tarjan and Valdes
Outline

- **What?**
  - Flow graph parsing

- **Why?**
  - Use cases for flow graph parsing

- **How?**
  - Our contribution: The Refined Process Structure Tree
  - Approach to computation: Using the triconnected components
  - An even more elegant algorithm
  - Lifting some assumptions
    - Dealing with separation points
    - Dealing with multiple sources/sinks
A Linear Time Algorithm for Computing the RPST (Overview)

Step 1: Detect the triconnected components.

Step 2: Analyze the triconnected components.

Step 3: Restructure the tree into the RPST.
Connectivity

- Undirected graph has a unique decomposition into *connected components*
Biconnectivity

- A *separation point* splits the graph into two or more components
- Splitting at all separation points yields a unique decomposition into *biconnected components*
Triconnectivity

- A *separation pair* splits the graph into two or more subgraphs (called *split graphs*).
- Entering/exiting a split graph is possible only via the nodes of the separation pair.
Fragments and Triconnectivity

- Add edge between sink and source ("return edge") and ignore directions
  - Resulting graph is w.l.o.g. biconnected
  - Each entry-exit pair (e.g. \((u,v)\)) of a non-trivial fragment is a separation pair
  - If \((u,v)\) is a separation pair, then the subgraph in between \(u\) and \(v\) is a fragment if \(u\) is an entry and \(v\) is an exit of that subgraph

- **Idea:** Find fragments through separation pairs and their split graphs
Triconnected Decomposition

- Always split into two split graphs
- Add a fresh edge $e$ between the separation pair to each split graph (called virtual edge)
- Recursive splitting of split graphs yields split components
- Split components are not necessarily unique
Non-determinism in splitting

- Decomposition along separation pairs is unique if polygons and bonds are not decomposed further → defines unique “triconnected components” (TCC)
- Triconnected components can be computed in linear time (Hopcroft and Tarjan 1973)
Step 1: Compute the Triconnected Components

A two-terminal graph

The undirected version

Component subgraphs

The tree of the triconnected components
Step 2: Analyze the Triconnected Components
Step 3: Restructure the Tree into the RPST

Step 3: Restructure the tree into the RPST.

Step 3 can be done in linear time
Examples of The Restructuring Step
Outline

- **What?**
  - Flow graph parsing

- **Why?**
  - Use cases for flow graph parsing

- **How?**
  - Our contribution: The Refined Process Structure Tree
  - Approach to computation: Using the triconnected components
  - An even more elegant algorithm
  - Lifting some assumptions
    - Dealing with separation points
    - Dealing with multiple sources/sinks
A more elegant computation (1/3)

- **Graph is** *normalized* if each node has either at most one incoming or at most one outgoing edge

- **Theorem**: For a normalized graph, RPST and Tree of TCC coincide
A more elegant computation (2/3) -- Idea

- Normalize graph by splitting nodes
- Compute Tree of TCC = RPST of normalized graph
- Project RPST back onto original graph
A more elegant computation (3/3)

1. Compute the Tree of the TCCs
2. Analyze the TCCs
3. Restructure the tree to the RPST

1. Split nodes
2. Compute the Tree of the TCCs
3. Remove added edges
4. Remove redundant fragments
Outline

- **What?**
  - Flow graph parsing

- **Why?**
  - Use cases for flow graph parsing

- **How?**
  - Our contribution: The Refined Process Structure Tree
  - Approach to computation: Using the triconnected components
  - An even more elegant algorithm
  - Lifting some assumptions
    - Dealing with separation points
    - Dealing with multiple sources/sinks
Dealing with Separation Points

- Use the same algorithm:
  - Normalize graph → separation points disappear
  - Compute RPST
  - Project RPST onto original graph

- Obtain fragments where entry and exit is the same node (e.g. P3)
Dealing with Separation Points

- Our decomposition creates natural hierarchy of fragments (a)-(c)

- Alternative:
  - Compute biconnected components
  - Decompose each biconnected component separately
  - Yields different, less natural decomposition, e.g. (d) vs. (c)
Dealing with Multiple Sinks

- Introduce dummy end node, then compute RPST
- Allows us to complete and refactor workflow graphs

\[ G_0 \]

\[ G_1 \]

\[ G_2 \]

\[ G_3 \]
Conclusion

- **Flow Graph Parsing** has
  - Many interesting use cases
  - **Requirements**: Uniqueness, modularity, granularity

- A new technique: **Refined Process Structure Tree**
  - Has simple characterization in terms of canonical fragments
  - Improves existing techniques by providing a **more fine-grained** decomposition
  - Unique, modular, can be **computed in linear time**

- Can be applied to other directed graphs (Petri nets, circuits)

- Implemented in IBM WebSphere Software
Backup Slides
Case Study: Soundness Checking

- State-space exploration:
  - Cannot analyze a process model having 100000 states
  → State space explosion
  • Considered intractable

- Divide-and-conquer technique:
  - Can check soundness of more process models
  - Can decrease the analysis time significantly if the state-spaces are large
    • E.g., by factor of 115
  - Can increase the analysis time if the state-spaces are small
    • E.g., by factor of 2.5

<table>
<thead>
<tr>
<th></th>
<th>State-space exploration</th>
<th>Divide-and-conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Library 1: 363 process models</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intractable</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Visited states</td>
<td>680559</td>
<td>2647</td>
</tr>
<tr>
<td>Total time</td>
<td>220 s</td>
<td>1.9 s</td>
</tr>
<tr>
<td><strong>Library 1: Excluding intractable</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visited states</td>
<td>80559</td>
<td>2647</td>
</tr>
<tr>
<td>Total time</td>
<td>11.3 s</td>
<td>1.9 s</td>
</tr>
<tr>
<td><strong>Library 2: 282 process models</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visited states</td>
<td>11793</td>
<td>2959</td>
</tr>
<tr>
<td>Total time</td>
<td>0.6 s</td>
<td>1.5 s</td>
</tr>
<tr>
<td>Intractable</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
A Linear Time Algorithm for Computing the RPST

Step 1: Detect the triconnected components.

Step 2: Check whether each triconnected component is a fragment.

Step 3: Restructure the tree into the RPST.
References

Reduction increases as the graph size increases

- Upper boundary for the largest fragment size seems to be independent of graph size
Triconnectivity

- A separation pair splits the graph into two or more subgraphs (called split graphs)
- A fresh edge $e$ between the separation pair is added to each split graph (called *virtual edge*)
- Recursive splitting of split graphs yields split components
Example

(a) 

(b) 

(c) 

(d)