University of Augsburg
Faculty for Distributed Systems

Business Process Validation

Documentation for the course
"'Distributed Systems Lab 2008"'

Manuel Majewski, Qiao Han, Armin Wurster
January 13, 2009

Abstract

Today creating workflows becomes more and more comfortable being supported by visual tools and editors. But how can we be sure that a created workflow is valid? It is quiet easy to recognize a small workflow as valid or invalid but if we have to handle complex workflows the validation might be an issue.

In this paper we introduce a algorithm based on the *fast heuristics* approach developed by IBM Research 2007 [1] to validate a workflow with linear effort und explain the theoretical backgroundknowlege the approach is based on. Further we show how this algorithm can be implemented as a plugin for the *JWT Workflow Editor* and discuss the results which were obtained using this algorithm to validate workflows generated with the *JWT Workflow Editor*.

1 Introduction

A correct execution of process models expects them to be modeled accurately. So, such a model has to be tested on certain qualities before one could start to execute it, e.g. on a process engine. Basic problems during executing a workflow are deadlocks, life locks or even circles, that interrupt the expected progress of actions within the workflow. To recognize and solve these problems, a *validation* of the workflow is required. The ambition of this study is to explain and implement a solution for process validation based on *fast heuristics* [1], facing the exponential complexity of validating larger process models.
This paper is structured as follows: In the second section we present the underlying equipment. The *JWT Workflow Editor* is explained as tool for simply modeling business processes. Furthermore, we describe the *Workflow Code Generation Framework* being a flexible implementation using Model Driven Architecture concepts to generate executable workflow code from an arbitrary process graph definition. It provides the groundwork for the validation of process models afterwards. The third section will give an overview how validation of workflows is done in general. Here we explain the requirements expected for the validation and how correctness of a workflow is verified. To sum up, we apply the previous theory of validation in the forth section pointing out its usage according to our concrete project, the validation of *JWT workflows*. Therefore, we describe our approach for matching the tools and requirements and extending the code to set up the validation tool. Finally, the concrete way of implementation is shown in the fifth section.

2 Tools and Frameworks

The project sets up on two basic tools, the *JWT Workflow Editor* and the *Workflow Code Generation Framework*. This section gives a short summary of their structure and functionality.

2.1 Workflow Editor

For process modeling, there exist several tools to achieve this purpose. This project is based on the *JWT Workflow Editor*, which is part of the Eclipse Java Workflow Tooling (JWT) Project. Basically, it allows users to create, manage and review workflow definitions within an appealing grafical user interface. This section is intended to give a little overview of the Workflow Editor, which is available as Eclipse Plugin, but also as RCP version (which is called AgilPro).
In order to model business processes in the Workflow Editor, it provides several visual elements that could be combined and form altogether the directed workflow graph. The most important ones are actually the *Activity Nodes*. They describe an activity within the workflow that could have a connection to each other. A subset of the *Activity Nodes* are the *Executable Nodes*, which could have references to some resources obtained from packages, the *Referencable Elements*. They could be one of the following:

- a role, responsible for the action
- an application, that is executed with the action
- some data, that is needed for the action

The *Referencable Elements* can be connected to the *Executable Nodes* by a special connection, the *Reference Edges*.
Hence, for finite graphs, a start and a end node is required. These can be found under another subset of the *Activity Nodes*, the *Control Nodes*, which also includes some nodes for obtaining more complex workflows.
The following *Control Nodes* are available:

- *Initial Node* and *Final Node*: Nodes that represent the begin and end of the workflow.
- *Fork Node* and *Join Node*: A *Fork Node* divides the flow requiring *all* outgoing paths to be executed, a *Join Node* recombines multiple paths to one flow.
- *Decision Node* and *Merge Node*: A *Decision Node* splits the flow executing only *one* of the outgoing paths, whereas a *Merge Node* takes exact one of multiple input paths to continue the flow. For special case, outgoing paths of a *Decision Node* can be attached by a *Guard*, which can carry forward the workflow in a certain direction.

To get a real activity flow, *Activity Edges* have to be set to connect between different *Activity Nodes*. Therefore, there are special rules for the different nodes, e.g. each *Executable Node* must have exactly one input and one output edge, but *Control Nodes* may have according their functionality $0..n$ *Activity Edges* on incoming or outgoing side. A combination of some nodes can be seen in Figure 1. Such a workflow model

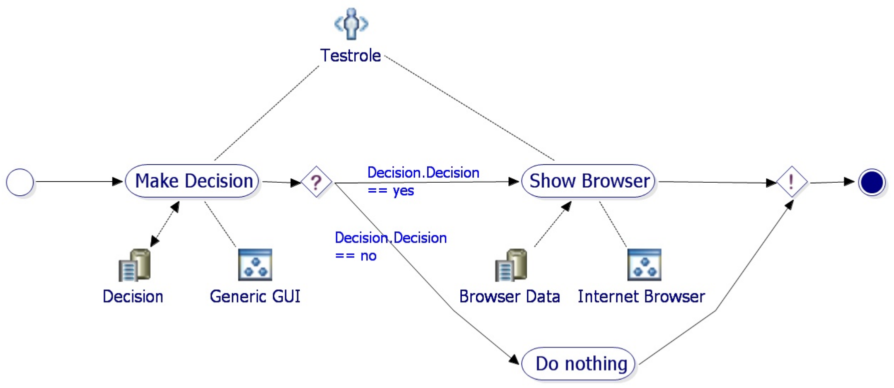


Figure 1: A sample workflow that shows an Initial Node, a Final Node, a Decision Node, a Merge Node and three Activity Nodes.

can now be taken for visualizing purposes, e.g. reminding a certain business process to employees, or even testing a course of business in a process engine. For a simple tutorial how to create a workflow, please have a look on this paper [9].

2.2 Workflow Generation Framework

The Workflow Generation Framework provides the basic circumstance for the generation of executable workflow code , for e.g. BPEL generation. It offers two mechanisms which enable an efficient code generation from any process models. They are the template mechanism and the adapter mechanism. The adapter mechanism allows the import of process models which are represented with any kind of process modeling format. And the template mechanism is responsible for code generations. The most important effort of this Framework is to separate components that depend on the domain and the execution environment from components that handle with computational tasks like control flow analysis and transformation. This improvement makes it possible, to reuse the components which are already integrated with the adapters, graph-transformation algorithms and code generation templates.

2.2.1 A structural view of the Workflow Generation Framework

From the structural aspect we may divide this Framework into four components. They are the *adapter for DSL process models (I)*, *process transformer and optimizer (II)*, the *process visitor (III)*, and the *code and model generation templates (IV)*. The relationship between these four components may be represented in Fig. 2.

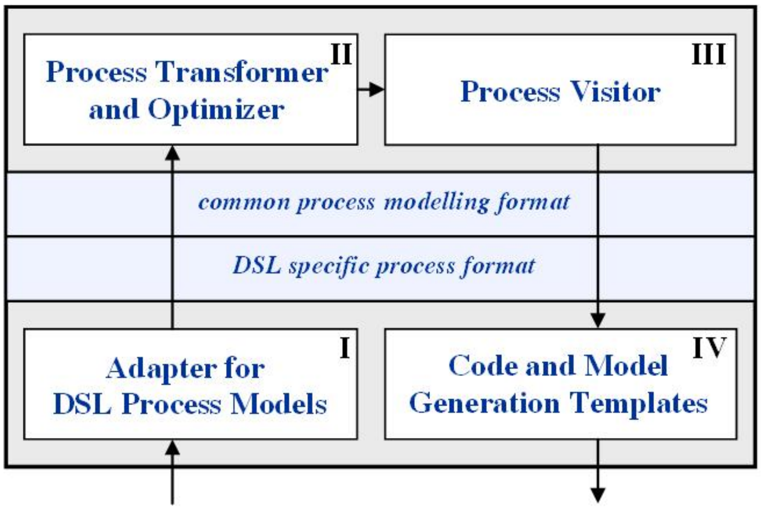

Process Transformer and Optimizer
II
Process Visitor
III
common process modelling format
DSL specific process format
Adapter for DSL Process Models
I
Code and Model Generation Templates
IV

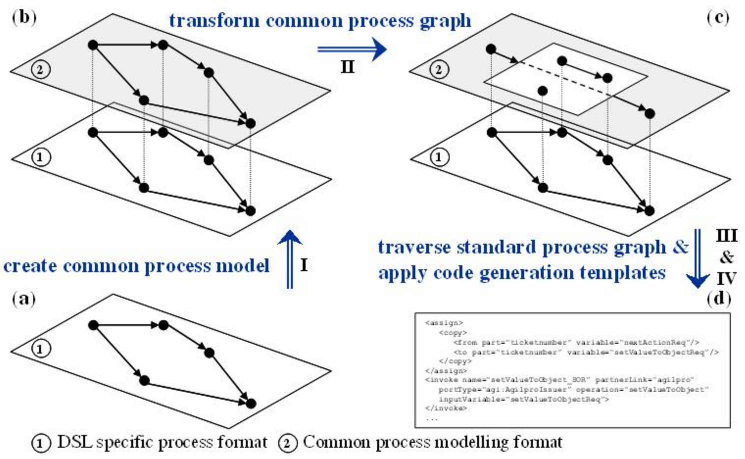
Figure 2: Model and Code Generation Framework

- The *process transformer and optimizer (II)* and the *process visitor (III)* are domain independent. They are in charge of issues during the general graph transformation and are aimed to transform a graph into every possible process modeling languages. The Process transformer is the most important component for code generation, it completes the transformation from the common process format into the format which can be further handled by the process visitor. Subsequently, the process visitor traverses the process model and then calls templates which will generate the workflow code. Another contribution of the visitor is to allow to generate the workflow code in the same sequence that is specified by the processes control flow.
- The *adapter for DSL process models (I)* and the *code and model generation templates (IV)* can directly access the workflow model which is modeled with any process modelling format. Because adapters and generation templates should always be used together, they must use the same DSL specific process format.

The *process transformer and optimizer (II)* and the combination of a visitor based and a template-based code generation contribute to a significant performance of the code generation. *Process transformer and optimizer* can furthermore identifiy Single-Entry-Single-Exit (SESE) fragments from process description. [10, 11 , 12]. The SESE fragments are generated through decomposition of the source process model and are usually substantially smaller then the original process. The graph transformations will be applied to SESE Fragments and that allows to generate block-structured (BPEL) code [10, 13]. We can also use SESE Fragments to verify the soundness of the process control flow within reasonable time [10, 14].

2.2.2 A functional view of the Workflow Generation Framework

The code generation is completed in four significant steps: input the process model, transform into common process modeling format, optimize the process model and output workflow code. Fig.3 shows the whole procedure (a)-(d) in detail.


(b)
transform common process graph
II
(c)
②
①
create common process model
I
traverse standard process graph &
apply code generation templates
III
&
IV
(a)
(d)
① DSL specific process format ② Common process modelling format

```
<assign>
   <copy>
      <from part="ticketnumber" variable="nextActionReq"/>
      <to part="ticketnumber" variable="setValueToObjectReq"/>
   </copy>
</assign>
<invoke name="setValueToObject_SOR" partnerLink="agilpro"
   portType="agi:AgilproIssuer" operation="setValueToObject"
   inputVariable="setValueToObjectReq">
</invoke>
...
```

Figure 3: Code Generation

a) In the first step a process model will be imported and this process is represented in a format specific to the DSL which is used for modeling the process.

b) In the second step we apply the *Adapter* on the input process. At first, the *Adapter* will access every component of the process and gather DSL specific information. According to this information the *Adapter* can create a new representation of the input process in the common process modeling format. Furthermore, it also sets a link between every processing step in the common format and the corresponding step of the input process. That ensures the traceability between the two process models.

c) The *process transformer* will now take over this new representation from the *Adapter*. During this step the process model will be further transformed and optimized. For example it may be transformed into a block-structured graph. Although this new representation may be different from the original one, with the aid of the links we can at any time reach every step of the both representations.

d) In the last step the process model will be traversed by the *visitor*, the *code and model generation templates* will be called by using the notification mechanism of the framework and the workflow code will be then generated.

3 Validation of Workflows

This section explains how the correctness of a *workflow* can be validated. First we show a *workflow* specified as a directed graph. Syntactic correct *workflow graphs* are called *sound*. For the validation of complex graphs the graph is decomposed into sub graphs called fragments, which are again *workflow graphs*.

3.1 Workflows

A *workflow* can be described as a directed graph $\mathcal{G}$ [A mp]. $\mathcal{G}$ is denoted by ($\mathcal{N}$,$\mathcal{E}$), where $\mathcal{E}$ is the set of edges and $\mathcal{N}$ the set of nodes. $\mathcal{N}$ consists of the disjoint subsets: $\mathcal{N}_{start}$, $\mathcal{N}_{stop}$, $\mathcal{N}_{executable}$, $\mathcal{N}_{fork}$, $\mathcal{N}_{join}$, $\mathcal{N}_{decision}$, $\mathcal{N}_{merge}$ so that
$\mathcal{N}_{start} \cup \mathcal{N}_{stop} \cup \mathcal{N}_{executable} \cup \mathcal{N}_{fork} \cup \mathcal{N}_{join} \cup \mathcal{N}_{decision} \cup \mathcal{N}_{merge} = \mathcal{N}$,
$\forall n \in \mathcal{N}$: $n \in (\mathcal{N}_{start} \vee \mathcal{N}_{stop} \vee \mathcal{N}_{executable} \vee \mathcal{N}_{fork} \vee \mathcal{N}_{join} \vee \mathcal{N}_{decision} \vee \mathcal{N}_{merge})$
and
$\mathcal{N}_{start} \cap \mathcal{N}_{stop} \cap \mathcal{N}_{executable} \cap \mathcal{N}_{fork} \cap \mathcal{N}_{join} \cap \mathcal{N}_{decision} \cap \mathcal{N}_{merge} = \emptyset$.
Each Node $n \in \mathcal{N}$ has a set of incoming and outgoing edges $\mathcal{E}_{in}(n)$, $\mathcal{E}_{out}(n)$. Further a *workflow graph* has to satisfy following conditions:

- $\mathcal{N}_{start}$ and $\mathcal{N}_{stop}$ have exactly one element (n_{start}, n_{stop}), such that ($\mathcal{E}_{in}(n_{start})$ $= \emptyset \wedge |\mathcal{E}_{out}(n_{start})| = 1$(called *entry edge* e_{entry})) $\wedge$ ($\mathcal{E}_{out}(n_{stop}) = \emptyset \wedge |\mathcal{E}_{in}(n_{stop})| =$ 1(called *exit edge* e_{exit})).
- $\forall n \in (\mathcal{N}_{fork} \vee \mathcal{N}_{decision})$: $|\mathcal{E}_{in}(n)| = 1 \wedge |\mathcal{E}_{out}(n)| >= 2$, $\forall n \in (\mathcal{N}_{join} \vee \mathcal{N}_{merge})$: $|\mathcal{E}_{in}(n)| >= 2 \wedge |\mathcal{E}_{out}(n)| = 1$ and $\forall n \in \mathcal{N}_{executable}$: $|\mathcal{E}_{in}(n)| = 1 \wedge |\mathcal{E}_{out}(n)| = 1$
- $\forall n \in \mathcal{N} : \exists$ a path p in $\mathcal{G}$, so that n, n_{start}, $n_{stop} \in p$

The graphical syntax of a is shown in figure 4 which displays a concrete example of a *workflow graph*.
A *workflow graph* $\mathcal{G}$ = ($\mathcal{N}$,$\mathcal{E}$) has a set of *states* denoted by $\mathcal{S}$. All *states* $s \in \mathcal{S}$ of $\mathcal{G}$ are defined by *tokens* on the edges of $\mathcal{G}$. Every *state* s of $\mathcal{G}$ is a relation $s : \mathcal{E} \rightarrow \mathbb{N}$, which maps a number $k \in \mathbb{N}$ to each edge $e \in \mathcal{E}$ hence $s(e) = k$ means that e carries k *tokens* in s.
Let s and $s' \in \mathcal{S}$ be two *states* and $n \in \mathcal{N}$ a node of $\mathcal{G}$, where $n \notin \mathcal{N}_{start}, \mathcal{N}_{stop}$.
So we define:

- $s \xrightarrow{n} s'$ if s changes to s' by executing n.
- n *activated* in s if $\exists$ a *state* s' such that $s \xrightarrow{n} s'$.
- $s \xrightarrow{*} s'$ (s is *reachable* from s) if $\exists$ a finite sequence *seq*: $s_0 \xrightarrow{n_1} s_1 ... s_{k-1} \xrightarrow{n_k} s_k$ ($k \in \mathbb{N}_0$) such that $s_0 = s$ and $s_k = s'$.

The set $\mathcal{S}$ of $\mathcal{G}$ includes the subsets $\mathcal{S}_{initial}$, $\mathcal{S}_{terminal}$ and $\mathcal{S}_{stopping}$, where $\mathcal{S}_{initial}$ contains all *initial states*, $\mathcal{S}_{terminal}$ contains all *terminal states* and $\mathcal{S}_{stopping}$ contains all *stopping states*. Because of $|\mathcal{N}_{start}| = 1 \wedge |\mathcal{N}_{stop}| = 1 \rightarrow |\mathcal{S}_{initial}| = 1 \wedge |\mathcal{S}_{terminal}|$ $= 1$.

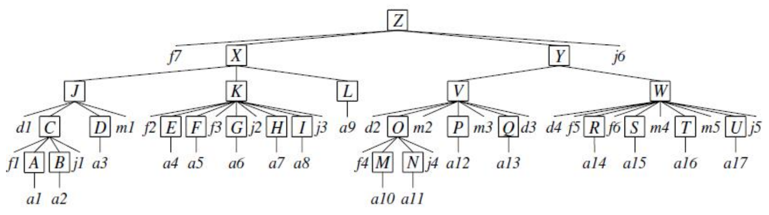
3.2 Sound Workflow Graphs

In this section we define the *soundness* [A mp,14] of a *workflow graph* $\mathcal{G}$. The *soundness* is an extensive term about correctness of a *workflow graph* hence relevant for the validation. $\mathcal{G}$ is *sound* if:

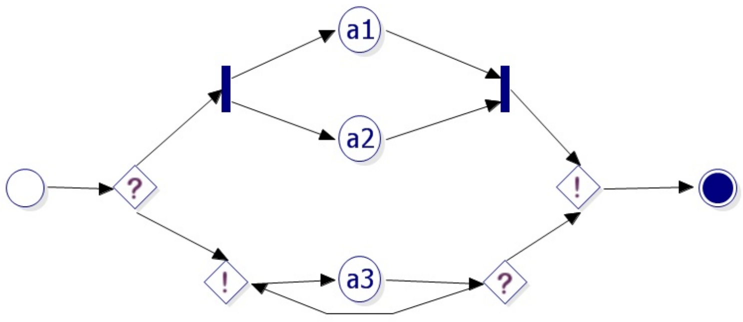
- $s_{initial} \in \mathcal{S}_{initial}$ of $\mathcal{G}$ has exactly one *token* t on e_{entry} and no *token* elsewhere.
- $s_{terminal} \in \mathcal{S}_{terminal}$ of $\mathcal{G}$ has exactly one *token* t on e_{exit} and no *token* elsewhere.
- $s_{stopping} \in \mathcal{S}_{stopping}$ of $\mathcal{G}$ implies that the set of *tokens* $\mathcal{T}$ derived from e_{exit} has at least one element.
- $\mathcal{G}$ is *live* and *safe*, where $\mathcal{G}$ is *live* describes that $\forall s \in \mathcal{S}$ reachable from $s_{initial}$ $\exists$ $s_{stopping} \in \mathcal{S}_{stopping}$ reachable from s and $\mathcal{G}$, is *safe* says that $\mathcal{S}_{stopping} \setminus \mathcal{S}_{terminal} = \emptyset$.

In figure 4 shows a *sound workflow graph* where the *workflow graphs* in figure 5 are both *unsound* because of *structural conflicts* (a *local deadlock* and a *lack of synchronization*) [A 11].

«««< .mine


Z
f7 X Y j6
J K L V W
d1 C D m1 f2 E F f3 G j2 H I j3 a9 d2 O m2 P m3 Q d3 d4 f5 R f6 S m4 T m5 U j5
f1 A B j1 a3 a4 a5 a6 a7 a8 f4 M N j4 a12 a13 a14 a15 a16 a17
a1 a2 a10 a11

=======



»»»> .r100

Figure 4: The graphical syntax of a concrete *workflow graph*. Executable nodes are shown as named circles, fork and join nodes as thin rectangles decision and merge nodes as diamonds with question- or explonitionmarks inside, start and stop nodes as (decorated) circles.

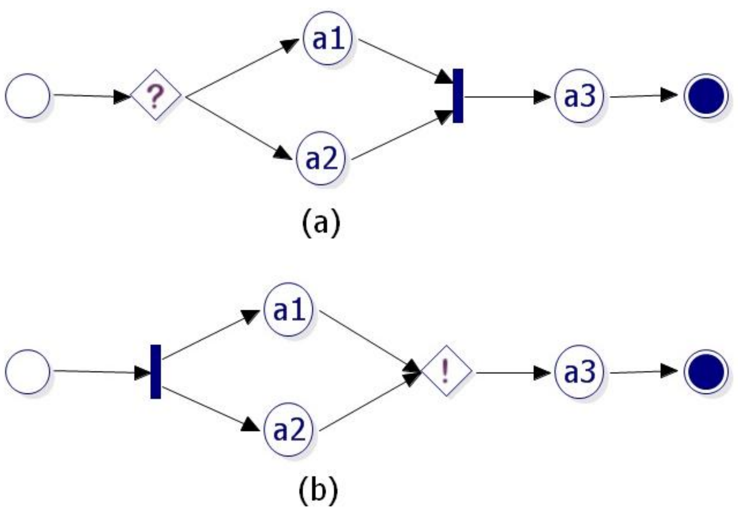


Figure 5: The *workflow graph* in (a) is not *live* (deadlock) where the *workflow graph* in (b) is not *safe* (lack of synchronization)

A *local deadlock* (a) is a *state* $s \in \mathcal{S}$ if $\exists$ a *state* $s' \in \mathcal{S}$, a node $n \in \mathcal{N}_{join}$, where $s_0 \overset{n}{\to} s_1 \overset{*}{\to} s_k$ $(k \in \mathbb{N}, s_0 = s, s_k = s')$ such that $\exists$ a incoming edge $e(n) \in \mathcal{E}_{in}(n)$ such that $e(n)$ carries a *token* in any *reachable state* s' of s and $\exists$ a incoming edge $e(n) \in \mathcal{E}_{in}(n)$ such that $e(n)$ does not carry a *token* in any *reachable state* s' of s.
A *lack of synchronization* (b) is a *state* $s \in \mathcal{S}$ if $\exists$ a node $n \in \mathcal{N}_{merge}$ such that $\exists$ more than one incoming edge $e(n) \in \mathcal{E}_{in}(n)$ carries a *token*.

We define a *workflow graph* $\mathcal{G}$:

- as locally live if $\nexists$ a *local deadlock* such that $s_0 \overset{*}{\to} s_k$ $(k \in \mathbb{N}, s_0 \in \mathcal{S}_{init}, s_k{=}\textit{local deadlock})$.
- as locally safe if $\nexists$ a *state* s such that $s_0 \overset{*}{\to} s_k$ $(k \in \mathbb{N}, s_0 \in \mathcal{S}_{init}, s_k = s$ and $\forall$ edges $e \in \mathcal{E}$: $s(e) <= 1$.

Theorem 1. *A workflow graph $\mathcal{G}$ is sound $\leftrightarrow$ $\mathcal{G}$ is locally safe and locally live.*

3.3 Enhanced Control-Flow Analysis

To validate *soundness* of a complex *workflow graph* as a whole is very exhaustive. In this section we introduce an approach how a *workflow graph* can be recognized as *sound* by dividing the graph into *single-entry-single-exit* (SESE) fragments and validate some fragments quickly as *sound* or *unsound*.
First we show the decomposition into SESE fragments. After that we explain how some fragments be discovered as *sound* or *unsound* [A mp].

3.3.1 Workflow Fragmentation

A SESE fragment is a not empty sub graph $\mathcal{F} = (\mathcal{N}', \mathcal{E}')$ of $\mathcal{G}$, where $\mathcal{G}$ is a *workflow graph*.
So we say $\mathcal{F}$ is a SESE fragment of $\mathcal{G}$ if $\mathcal{N}' \subseteq \mathcal{N} \wedge \mathcal{E}' = \mathcal{E} \cap (\mathcal{N}' \times \mathcal{N}')$ such that $\exists$ a edge $e, e' \in \mathcal{E}$: $\mathcal{E} \cap ((\mathcal{N} \backslash \mathcal{N}') \times \mathcal{N}') = \{e\} \wedge \mathcal{E} \cap (\mathcal{N}' \times (\mathcal{N} \backslash \mathcal{N}')) = \{e'\}$, where $e = e_{entry}$

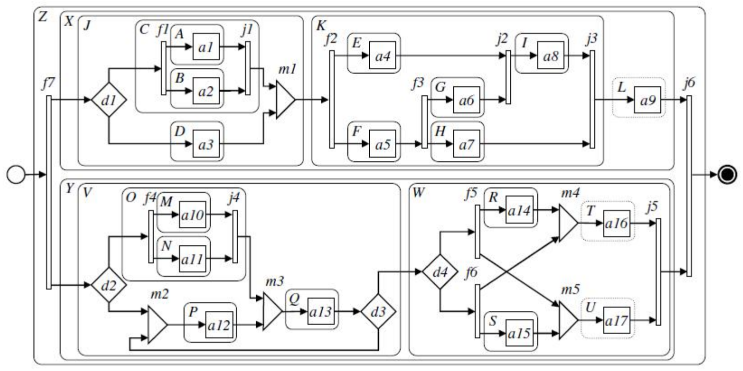


Figure 6: A *workflow graph* and its decomposition into SESE fragments. A SESE fragment is displayed as a dotted box. All fragments are *canonical.*

and $e' = e_{exit}$ of $\mathcal{F}$. A SESE fragment $\mathcal{F}$ can be considered as a *workflow graph* if we add the edges e_{entry}, e_{exit} and the nodes n_{start}, n_{stop}.

Canonical Fragments Two SESE fragments $\mathcal{F}, \mathcal{F}'$ are *in sequence* if e_{exit} of $\mathcal{F} = e_{entry}$ of $\mathcal{F}'$ or e_{exit} of $\mathcal{F}' = e_{entry}$ of $\mathcal{F}$. $\mathcal{F} \cup \mathcal{F}'$ describes a SESE fragment again. A fragment $\mathcal{F}$ is not *canonical* if $\exists$ fragments X, Y such that X and Y are *in sequence* $\wedge$ $\mathcal{F} = X \cup Y$ $\wedge$ $\exists$ a fragment Z such that $\mathcal{F}$ and Z are *in sequence*, else $\mathcal{F}$ is considered as *canonical.*
If we say fragment we only mean the particular fragments of a *workflow graph*, which are *canonical* in the following.
In figure 6 a *workflow graph* fragmented into *canonical* SESE fragments is shown [A mp,5].

3.3.2 Sound Fragments

Based on the definitions given by Hauser et al [A 4] we now introduce the conditions for a fragment to be recognized as *sound.* First we explain some general fragment definitions for a fragment $\mathcal{F}$ of a *workflow graph* $\mathcal{G}$: *well-structured, unstructured concurrent, unstructured sequential* and *complex.*
well-structured $\rightarrow$

- $\forall$ nodes $n \in \mathcal{N}$ of $\mathcal{F}$: $n \notin \mathcal{N}_{decision}, \mathcal{N}_{merge}, \mathcal{N}_{fork}, \mathcal{N}_{join}$.
- $|\mathcal{N}_{decision}| = 1$, $|\mathcal{N}_{merge}| = 1$, $\mathcal{N}_{fork} = \emptyset$ and $\mathcal{N}_{join} = \emptyset$ of $\mathcal{F}$. $e_{entry} = e_{in}(n_{decision}) \wedge e_{exit} = e_{out}(n_{merge})$.
- $|\mathcal{N}_{decision}| = 1$, $|\mathcal{N}_{merge}| = 1$, $\mathcal{N}_{fork} = \emptyset$ and $\mathcal{N}_{join} = \emptyset$ of $\mathcal{F}$. $e_{entry} = e_{in}(n_{merge}) \wedge e_{exit} = e_{out}(n_{decision})$.
- $|\mathcal{N}_{fork}| = 1$, $|\mathcal{N}_{join}| = 1$, $\mathcal{N}_{decision} = \emptyset$ and $\mathcal{N}_{merge} = \emptyset$ of $\mathcal{F}$. $e_{entry} = e_{in}(n_{fork}) \wedge e_{exit} = e_{out}(n_{join})$.

unstructured concurrent $\rightarrow$ $\mathcal{F} \neq$*well-structured*, contains no cycles and $\mathcal{N}_{decision} = \emptyset$, $\mathcal{N}_{merge} = \emptyset$.
unstructured sequential $\rightarrow$ $\mathcal{F} \neq$*well-structured* and $\mathcal{N}_{fork} = \emptyset$, $\mathcal{N}_{join} = \emptyset$.
complex $\rightarrow$ $\mathcal{F} \neq$*well-structured*$\wedge \neq$*unstructured concurrent*$\wedge \neq$*unstructured sequential.*

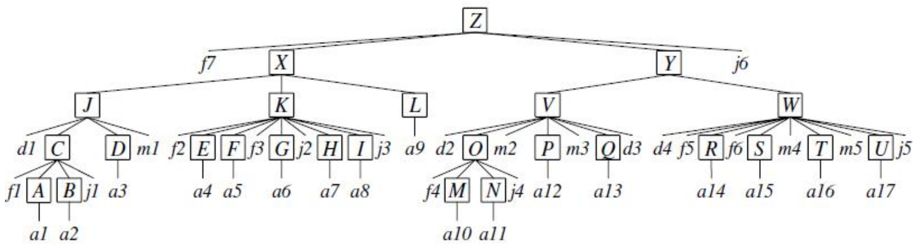

Z
f7 X Y j6
J K L V W
d1 C D m1 f2 E F f3 G j2 H I j3 a9 d2 O m2 P m3 Q d3 d4 f5 R f6 S m4 T m5 U j5
f1 A B j1 a3 a4 a5 a6 a7 a8 f4 M N j4 a12 a13 a14 a15 a16 a17
a1 a2 a10 a11

Figure 7: The *process structure tree* of a *workflow graph* which is shown in Figure X. The fragments are depicted as boxes and the leafs of the tree are the nodes of the graph.

The following theorem defines the *soundness* of a fragment using the definitions above.

Theorem 2. *$\forall$ $\mathcal{F}$ in $\mathcal{G}$: $\mathcal{F}$ is well-structured, unstructured concurrent or unstructured sequential $\rightarrow$ $\mathcal{F}$ is sound $\leftrightarrow$ $\forall$ $\mathcal{F}'$: $\mathcal{F}'$ is sound, where $\mathcal{F}'$ is child fragment of $\mathcal{F}$.*

3.3.3 Unsound Fragments

Fragments can be recognized as *unsound* if a *complex* fragment applies to theorem 3.

Theorem 3. *A fragment $\mathcal{F}$ is not sound $\rightarrow$ $\mathcal{F}$ is complex and one of the following conditions is satisfied:*

- $|\mathcal{N}_{merge}| > 0$ *(*$|\mathcal{N}_{decision}| > 0$*),* $\mathcal{N}_{decision} = \emptyset$ *(*$\mathcal{N}_{merge} = \emptyset$*) of* $\mathcal{F}$*.*
- $|\mathcal{N}_{fork}| > 0$ *(*$|\mathcal{N}_{join}| > 0$*),* $\mathcal{N}_{join} = \emptyset$ *(*$\mathcal{N}_{fork} = \emptyset$*) of* $\mathcal{F}$*.*
- $\mathcal{N}_{decision} = \emptyset$*,* $\mathcal{N}_{merge} = \emptyset$ *of* $\mathcal{F}$ *and* $\mathcal{F}$ *contains no cycle*

3.4 Sound Workflow Graphs

Two fragments of a *workflow graph* $\mathcal{G}$ are either nested or disjoint thus the decomposition into fragments can be ordered as a tree. Such a tree is called *process structure tree* of a $\mathcal{G}$ which is shown for the graph in figure 7 [A mp,5]. We denote a fragment $\mathcal{F}$ from a node $n \in \mathcal{N}$ of $\mathcal{G}$ $\mathcal{F}(n)$. We call a fragment $\mathcal{F}$ as *child fragment* of $\mathcal{F}'$ if $\mathcal{F}'$ contains $\mathcal{F}(n)$. We also define $\mathcal{F}'$ as the *parent fragment* of $\mathcal{F}$. A *workflow graph* $\mathcal{G}$ is recognized as *sound* if all fragments of $\mathcal{G}$ are *sound*.

Theorem 4. *A workflow graph $\mathcal{G}$ is sound $\leftrightarrow$ $\forall$ fragments $f \in$ child fragments: f is sound $\wedge\exists$ a sound workflow graph $\mathcal{G}'$, where $\forall$ child fragments $f \in \mathcal{G}$: $f \in \mathcal{N}_{executables}$ of $\mathcal{G}$ and otherwise $\mathcal{G}'$ is structured like $\mathcal{G}$ [A mp,12,13,14,18]*

To validate the *soundness* of a *A workflow graph* $\mathcal{G}$ we check the *soundness* of each fragment f of the *process structure tree* of $\mathcal{G}$ starting at the leafs and moving upwards to the root because if f is recognized as *sound* it can be considered as *executable node* in the *parent* fragment which is shown in figure 8.

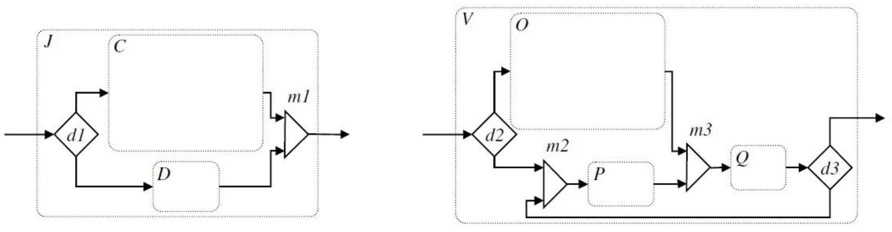


Figure 8: J and V are fragments of a *workflow graph* from figure X and X which ignore the structure of their *child fragments*

4 JWT Workflow Validation

This section explains how a *workflow graph* created in the JWT Workflow Editor, which has been introduced in section 2.1, can recognized as *sound* or *unsound*. The correctness of the *workflow graph* is validated applying the approach explained in section 3. Therefore it is necessary to decompose the *workflow graph* into its *canonical fragments* which are representing the nodes in the corresponding *process structure tree*. Now we can perform the validation of the *workflow graph* in linear time, which is the final goal of this paper.

4.1 Input graph

To perform a validation of *soundness* of a *workflow graph* $\mathcal{G}$ we need an editor which provides a concrete input graph. In this paper we show how *workflow graphs* created by the JWT Workflow Editor are validated.
The graph $\mathcal{G}$ provided by the JWT Workflow Editor is too concrete, thus it has to be reduced to a *workflow graph* $\mathcal{G}_{red}$ defined in section 3. Information about guards, etc. is not needed to recognize a *workflow graph* as *sound* or *unsound*, so only *Executable Nodes* and *Control Nodes*, including their connections, the *Activity Edges*, are required. The adapter of the *Codegen Framework* implements the reduction from $\mathcal{G}$ to $\mathcal{G}_{red}$ [codegen framework adapter].

4.2 Fragmentation into canonical fragments

In order to derive the *canonical fragments* $\mathcal{F}$ from a *JWT Workflow Editor* graph $\mathcal{G}$ we use the adapter provided by the *Workflow Codegen Framework* to reduce $\mathcal{G}$ into a more abstract graph $\mathcal{G}_{red}$, where specific information of $\mathcal{G}$ has been removed. Actually, we change or remove basically the following elements from the input graph $\mathcal{G}$:

- $\forall n \in \mathcal{N}_{process}$ of $\mathcal{G}$: $n \in \mathcal{N}_{executable}$ of $\mathcal{G}_{red}$.
- the set of *Guards* in $\mathcal{G}$ will be removed in $\mathcal{G}_{red} \rightarrow$ the set of *Guards* in $\mathcal{G}_{red} = \emptyset$.
- $\mathcal{N}_{referencable}$ of $\mathcal{G}$ will be removed in $\mathcal{G}_{red} \rightarrow \mathcal{N}_{referencable} \cap \mathcal{G}_{red} = \emptyset$.

As result, there is a more abstract graph that can be further used within the *Codegen Framework.* Furthermore, we decompose this graph into a set of SESE fragments using the transformer. During this decomposition, we build up the *process structure tree* which is defined in section 3.

4.3 Validation

The *process structure tree* $\mathcal{T}_G$ is derived from a workflow graph $\mathcal{G}_T$, where each node $n \in \mathcal{N}$ of $\mathcal{T}_G$ is considered as a fragment $\mathcal{F} \in \mathcal{G}_T$ defined in section 3. $\forall n \in \mathcal{N}$ of $\mathcal{T}_G$: n is a *canonical fragment* $\mathcal{F}_c$.
In our case these features are provided by the *tokenanalysis framework* so we only have to put all nested or disjoint fragments $\mathcal{F}_c$ together to a *process structure tree* $\mathcal{T}_G$. All fragments $\mathcal{F}'_c$ nested in $\mathcal{F}_c$ become child fragments of $\mathcal{F}_c$. We recognize $\mathcal{F}_c$ as *sound* or *unsound* as we defined *soundness* in section 3. If $\mathcal{F}_c$ is *unsound* we mark $\mathcal{F}_c$ in order to remember that $\mathcal{F}_c$ has been modeled incorrectly. Note that the fragment will be just marked, the validation continues ignoring the result of the fragment validation. Hence more potential *unsound* fragments can be located in $\mathcal{T}_G$. Further we do not have to care about the way we validate the fragments $\mathcal{F}_c \in \mathcal{T}_G$. The only condition is that all fragments $\mathcal{F}_c \in \mathcal{T}_G$ have to be validated if the algorithm terminates.
Whenever a fragment $\mathcal{F}_c \in \mathcal{T}_G$ occurs as a child fragment of $\mathcal{F}'_c \in \mathcal{T}_G$ $\mathcal{F}_c$ will be replaced by an *Executable Node* $n_{executable}$ in $\mathcal{F}'_c$. If all nodes $n \in \mathcal{N}$ of $\mathcal{T}$ are marked as *sound* or *unsound* the output of the validation of $\mathcal{G}$ will be generated by refering each *unsound* fragment to thier origin in $\mathcal{G}$. Note that all nodes $n \in \mathcal{N}$ of $\mathcal{T}$ have direct connections to the *process structure tree* $\mathcal{T}_G$ which also is directly connected to $\mathcal{G}$. So each *unsound* fragment of $\mathcal{T}$ can easily be traced back to its origin in $\mathcal{G}$.

5 JWT Workflow Validation Plugin

5.1 An Overview

This Plugin provides a comfortable approach to validate process models. It allows the user to validate an existent process model or to create his own process model by using this Plugin and then to validate it. After the validation the unsound SESE fragments will be marked. The user may be allowed to adjust them and then lets the Plugin validate this new process model again.
The significant components of the *JWT Workflow Validation Plugin* are displayed in Fig. 9.
Because of setting up basically on the Codegen Framework, the base structure is very

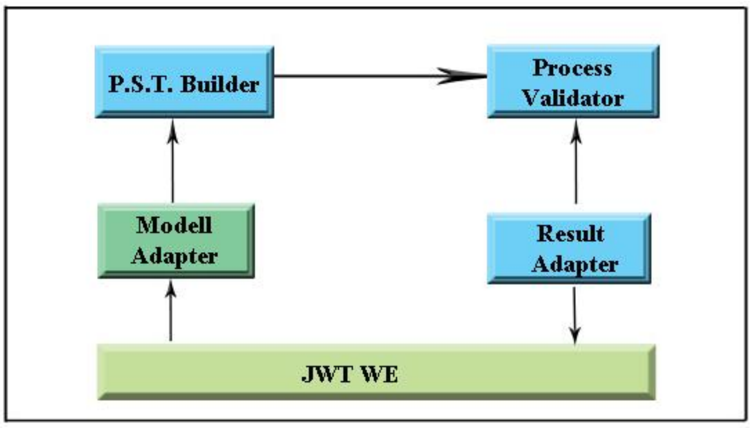


Figure 9: JWT Workflow Validation Plugin

similar, thus not creating code as output, but displaying the results in the Workflow

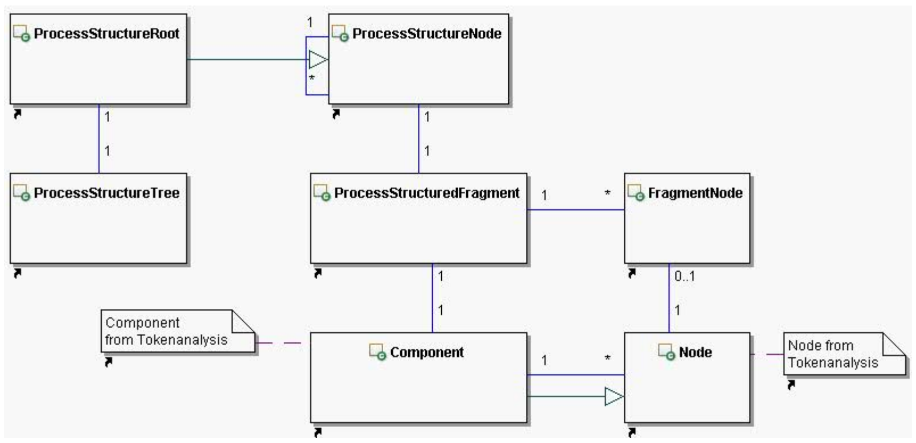


Figure 10: The classdiagramm of the implementation of the *process structure tree*.

Editor. As first step, we used the *Model Adapter* from the Codegen Framework to get access to the graph model of the Workflow Editor. It provides the common process model format we wanted to operate on for validation. Next, our first implemented component is the *Process Structure Tree Builder*, which transforms the abstract graph representation into single-entry-single-exit fragments. The chosen result structure is a tree of each node point to a fragment, which each has enough information to be validated. This is the input for the *Process Validator*, which actually validates each fragment based on the rules we described in section 3. The validation gives as result set the lists of sound, unsound and undetermined fragments. The last component, the *Result Adapter* takes these lists and offers some methods to handle the result, e.g. for visualization in the Workflow Editor. In the following subsections, our implementation of the new components is described in details.

5.2 Process Structure Tree Builder

The components provided by *tokenanalysis framework* represent the *canoncial fragments* we need for the validation. The components consist of a set of component nodes which represent the structure of the fragment.
So we just need to put the fragments which are either nested or disjoint into a tree format. The class structure of the *process structure tree* is shown in figure 10.
The *process structure tree* consists of a set of nodes, where each node contains a fragment. This fragment may contain other child fragments. The *ProcessStructureTreeFragments* refer to the basic fragments so that each *ProcessStructureTreeFragment* refers to a component and each *FragmentNode* refers to the according component node. Additionally the *FragmentNode* provide the reference to the original node of the *workflow editor* workflow and the node type information which is important for the validation.
The *ProcessStructureTreeBuilder* acts as a transformer which transforms a workflow-graph created by the *workflow editor* which is already decomposed into fragments by the *tokenanalysis framework* into the *process structure tree structure* introduced above and shown in figure 10 and ensures that each node in each fragment can be traced back to the curresponding node of the BP_Meta_Graph. The BP_Meta_Graph

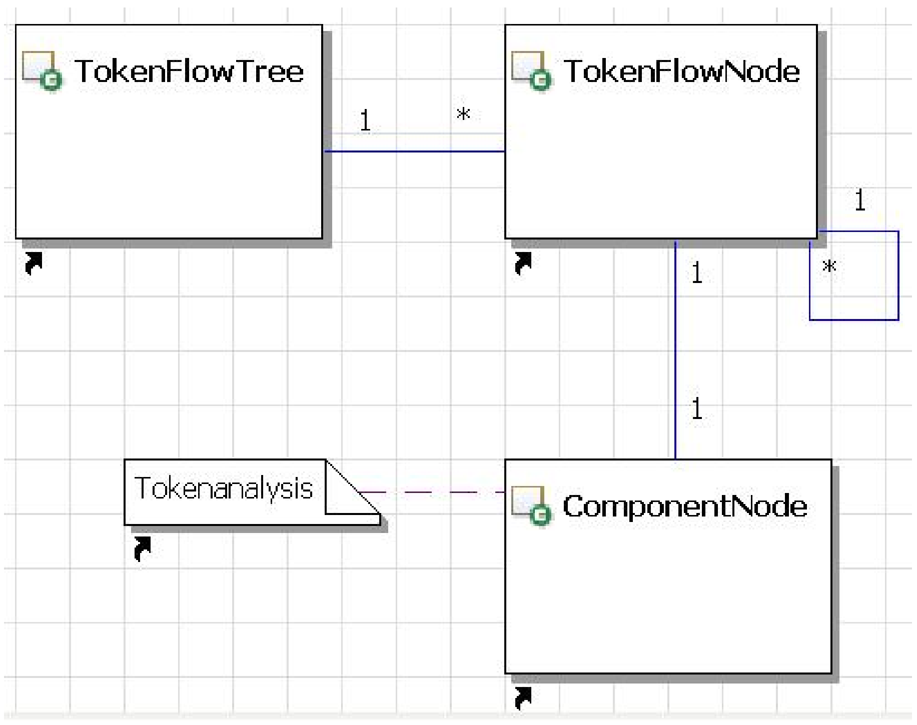


Figure 11: The classdiagramm of the implementation of the *tokenflow tree*.

represents the source workflow graph for the transformer.

5.3 Tokenflow Tree Builder

This transformer is only used for *acyclic complex fragments* which can not be recognized as *sound* by *Theorem 3.*. This component returns a set of *tokenflow trees*, where each tree can be considered as a possible tokenflow through the complex fragment. The class diagramm of the *tokenflow tree structure* is shown in figure 11. Similar to the *process structure tree* introduced in 5.2 the *tokenanalysis framework* provides the component nodes representing the structure of the fragment. Based on this structure the *tokenflow trees* were derived from the *FragmentNode* which containing fragment is regcognized as an *acyclic complex fragment.*

5.4 Process Validator

An essential component of this Plugin is the *Process Validator* which implements the theory of this article to validate the SESE fragments from the *Process Structure Tree.* The *Process Validator* checks at first whether a SESE fragment can be verified to be a *well-structured*, *unstructured-concurrent* or unstructured-sequential fragment. In case the SESE fragment matches one of them, it is validated to be *sound* and the *Process Validator* begins to validate the next one. Otherwise the *Process Validator* checks whether this SESE fragment matches one of the conditons which are mentioned in the *Theorem 3.* In case this SESE fragment satisfies the *Theorem 3*, it is validated as an *unsound fragment.* Otherwise it is undetermined, that is to say, this SESE fragment is neither *sound* nor *unsound.* The *Process Validator* works on every SESE fragment

one by one, until the whole *Process Structure Tree* is finished with validation.
The validation of *complex fragments* which are not recognized as *unsound* using the *fast heuristics* algorithm remain *undecided.* The subset of these fragments which are *acyclic* can be validated by validating all responding *tokenflow trees* generated by the *TokenFlowTreeBuilder* for each fragment. The conditions for a *tokenflow tree* to be recognized as *sound* are explained in **??**.
The *TokenFlowResult* component is used to validate each *tokenflow tree.* Note that the *unsoundness* of only one *tokenflow tree* of the set of *tokenflow trees* generated from the *complex fragment* causes the *unsoundness* of the whole fragment.

5.5 Validation Result Adapter

The aim of the *Result Adapter* is to handle the result set that's given by the *Process Validator* for a concrete modeller, that is in this case the Workflow Editor. It takes the list of sound, unsound and not determined fragments, but also the list of formal invalid fragments and marks them within the editor. Unsound fragments are marked with red color as well as formal invalid nodes (nodes that are missing an edge for validation). Not determined fragments will be set to a green color. So, one could see fast what's wrong with the tested workflow. For unsound and not determined fragments, there is set a log entry for notice which fragment has failed validating and what's the reason for that.
Additionally, we've implemented a method to the adapter that shows the results as message box on the screen, so you could see all nodes of each sound, unsound or not determined fragment, but also the count of formal invalid nodes.
Sum up, the *Result Adapter* provides all information the user can see after validating the workflow.

6 Conclusion

In this work we introduced a visual tool for creating workflow process, JWT Workflow Editor.This Editor makes the creation of workflow process simple and comfortable.The created process can also be stored for further use. We have also provided an Introduction to the Workflow Generation Framework from both the structural aspect and the functional aspect. The assignment of this Framework is to generate workflow code from any kind of process models. In order to evaluate process models we introduced the definition of soundness. Furthermore we gave a detailed description for determining the soudness of a process model. At last we proposed and implemented a concrete tool to validate workflow process. This tool is implemented as a Eclipse Plug-in and is an important part of the project Eclipse Java Workflow Tooling (JWT).
Our main goal, to validate a workflow within linear effort was not obtained here. The implementation of the *fast heuristic* algorithm lacks of completeness, because there is a subset of possible fragments we can not be validated with this approach. Thus we implemented an other approach to validate the *complex fragments* which can not be recognized as *sound* or *unsound* using the *fast heuristic* algorithm. The second algorithm lacks of a general proof of corectness and also of completeness, because only *acyclic complex fragments* can be validated. Moreover the effort it takes to validate a *acyclic complex fragments* is expotential not linear. So there there is still the subset of *cyclic complex fragments* left, which still remain *undecided.*
In the future there may be some refinements for the implementation for the *fast heuristic* algorithm. But the main issue to be solved is that the whole possible set of *complex fragments* could be validated (with linear effort?).
SO plugin for workflow validation desribed within this paper is just the first step providing validation tool support for the *JWT Workflow Editor.*

Acknowledgments
We thank Florian Lautenbacher for supplying this opportunity and his valuable suggestions. We also thank Benjamin Honke, who implemented the Workflow Generation Framework, and the chair for Programming Distributed Systems at the University of Augsburg.

Literatur

[1] Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition Jussi Vanhatalo, Hagen Völzer1, and Frank Leymann; IBM Zurich Research Laboratory; Springer-Verlag Berlin Heidelberg 2007

[2] van der Aalst,W.M.P., Hirnschall, A. (Eric) Verbeek, H.M.W.: An alternative way to analyze workflow graphs. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 535 to 552. Springer, Heidelberg (2002)

[3] Sadiq,W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. 25(2), 117 to 134 (2000)

[4] Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In: PLDI. Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pp. 171 to 185. ACM Press, New York (1994)

[5] Hauser, R., Friess, M., K¨uster, J.M., Vanhatalo, J.: An incremental approach to the analysis and transformation of workflows using region trees. IEEE Transactions on Systems, Man, and Cybernetics Part C (June 2007) (to appear, also available as IBM Research Report RZ 3693)

[6] Valette, R.: Analysis of Petri nets by stepwise refinements. Journal of Computer and System Sciences 18(1), 35 to 46 (1979)

[7] van der Aalst, W.M.P.: Workflow verification: Finding control flow errors using Petrinetbased techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161 to 183. Springer, Heidelberg (2000)

[8] Zerguini, L.: A novel hierarchical method for decomposition and design of workflow models. Journal of Integrated Design and Process Science 8(2), 65 to 74 (2004)

[9] Using Token Analysis to Transform Graph-Oriented Process Models to BPEL; Götz, Roser, Lautenbacher and Bauer, Report 2008 08, Juni 2008

[10] de.uniAugsburg.wf-codegen.userManual_1.1.0.pdf; Benjamin Honke, Stephan Roser

[11] Mathias Götz, Stephan Roser, Florian Lautenbacher, and Bernhard Bauer. Using Token Analysis to Transform Graph-Oriented Process Models to BPEL. 2008, forthcoming.

[12] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In Conference on Programming language design and implementation, pages 171 to 185. ACM Press, 1994.

[13] Chun Ouyang, Marlon Dumas, Arthur H.M. ter Hofstede, and Wil M.P. van der Aalst. Pattern-based translation of BPMN process models to BPEL web services. International Journal of Web Services Research (JWSR), 2007.

[14] Jussi Vanhatalo, Hagen V¨olzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In 5th ICSOC Conference, LNCS, pages 43 to 55. Springer, 2007.)