Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

On Similarities Between Workflow Verification and Grammar Checking

Roman Barták and **Vladislav Kuboň**
Charles University in Prague, Faculty of Mathematics and Physics,
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
bartak@ktiml.mff.cuni.cz, vk@ufal.mff.cuni.cz

Abstract

The paper investigates the similarities in the application of attribute grammars to two seemingly different research areas, namely the area of formal description of workflows and the area of checking the syntactic correctness of natural languages. It uses existing models and formalisms and tries to find a common ground which would enable to exploit mutually the experience gained in both individual fields. It shows how a slight adaptation of a grammar formalism used for grammar checking of languages with a high degree of word-order freedom may lead to a tool useful for a workflow verification.

Introduction

Although the description of workflows and the syntactic analysis of natural languages seem to constitute very distant research topics, the closer look at the nature of the tasks being solved reveals interesting similarities. If we look at them from a more abstract point of view, we may notice that they both deal with large units which may be decomposed into smaller units (the workflows into individual tasks and activities, the natural language sentences into clauses and individual words) and which have certain inner structure.

Workflows are used to formally describe processes of various types such as business and manufacturing processes. There exist many formal models to describe workflows (van der Aalst and t. Hofstede 2005) that include decision points and conditions for process splitting as well as loops to describe repetition of activities. Hierarchical structure of workflows is in particular interesting for real-life workflows (Bae et al. 2004) as many workflows are obtained by decompositions of tasks. Barták and Čepek (Barták and Čepek 2008) proposed a hierarchical workflow model called Nested Temporal Networks with Alternatives that was later extended with extra constraints to model a wider range of workflows (Barták et al. 2011). Later on, it has been shown that nested workflows with extra constraints can be represented by attribute grammars (Barták 2016), which constitute grounds to exploiting grammar-related techniques in workflow processing. For example, the problem of verifying whether a given word belongs to the language of a given grammar seems intrinsically similar to verify whether a specific process complies with the description of a workflow using an attribute grammar.

Natural language sentences seem to be even more complicated, they are less formal than workflows, more ambiguous and vague. On top of that, their correctness, as it is perceived by humans, does not depend entirely on syntax, it is very often also influenced by semantics, or, even worse, by a real-world knowledge. This broad perception of correctness must nevertheless be abandoned in applications such as automatic grammar checkers which may rely solely on relatively well defined syntactic rules. One attempt to build a pilot implementation of a grammar checker for a language with relatively high degree of word-order freedom has been described in (Holan, Kuboň, and Plátek 1997).

This paper investigates one issue which might bring together nested workflows and attribute grammars handling grammar checking of natural languages. We are going to describe how a concrete type of an attribute grammar might be used for verification of concrete workflows and what kind of modifications this grammar requires in order to be able to fulfill this task.

Nested Workflows

In this work we use nested workflows from the FlowOpt system (Barták et al. 2011). The nested workflows were formally introduced in (Barták and Rovenský 2014) and for completeness, we will recapitulate the formal definitions here.

The *nested workflow* is obtained from a root task by applying decomposition operations that split the task into subtasks until primitive tasks, corresponding to activities, are obtained. Three decomposition operations are supported, namely parallel, serial, and alternative decompositions. Figure 1 gives an example of a nested workflow that shows how the tasks are decomposed. The root task $Chair$ is decomposed serially to two tasks, where the second task is a primitive task filled by activity $Assembly$. The first task $Create\ Parts$ decomposes further to three parallel tasks $Legs$, $Seat$, and $Back\ Support$. $Back\ Support$ is the only example here of alternative decomposition to two primitive tasks with Buy and $Welding$ activities ($Welding$ is treated as an alternative to Buy). Hence the workflow describes two alternative processes. Naturally, the nested workflow can be

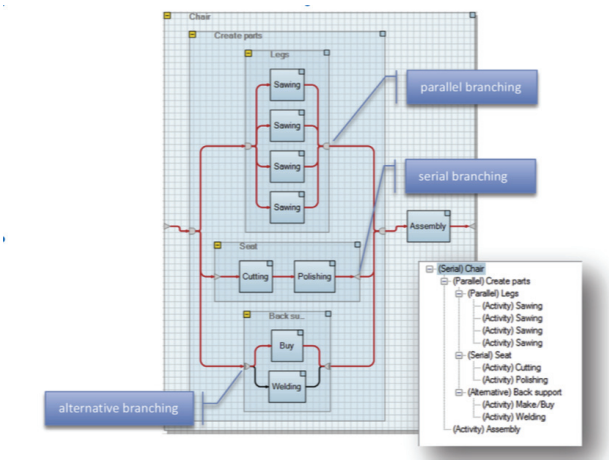

Figure 1: Example of a nested workflow as it is visualized in the FlowOpt Workflow Editor (from top to down there are parallel, serial, and alternative decompositions).

described as a tree of tasks (Figure 1 bottom right).

Formally, the nested workflow is a set $Tasks$ of tasks that is a union of three disjoint sets: $Parallel$, $Alternative$, and $Primitive$. For each task T (with the exception of the $root$ task), function $parent(T)$ denotes the parent task in the hierarchical structure. Similarly for each task T we can define the set $subtasks(T)$ of its child nodes ($subtasks(T) = \{C \in Tasks | parent(C) = T\}$). The tasks from sets Parallel and Alternative are called *compound tasks* and they must decompose to some subtasks:

$$\forall T \in (Parallel \cup Alternative) : subtasks(T) \neq \emptyset, \quad (1)$$

while the *primitive tasks* do not decompose:

$$\forall T \in Primitive : subtasks(T) = \emptyset. \quad (2)$$

The workflow defines one or more processes in the following way. *Process* selected from the workflow is defined as a subset $P \subseteq Tasks$ in the workflow satisfying the following constraints:

$$\forall T \in P, T \neq root : \quad parent(T) \in P \quad (3)$$
$$\forall T \in P \cap Parallel : \quad subtasks(T) \subseteq P \quad (4)$$
$$\forall T \in P \cap Alternative : \quad |subtasks(T) \cap P| = 1 \quad (5)$$

Formula (3) says that for each task in the process (except the root) its parent task is also in the process. Formula (4) says that all subtasks of a parallel task in the process must also be in the process. Finally, formula (5) says that exactly one subtask is in the process for each alternative task in the process.

In addition to the hierarchical structure of the nested workflow, the nested structure also defines certain implicit temporal (ordering) constraints (the arcs in Figure 1). These temporal relations must hold for all tasks in a single process. Assume that S_i is the start time and E_i is the end time of task T_i. The primitive tasks T_i are filled with activities and each activity has certain duration D_i. Then for tasks in the process P the following relations hold:

$$\forall T_i \in P \cap Primitive : S_i + D_i = E_i \quad (6)$$
$$\forall T_i \in P \cap (Parallel \cup Alternative) :$$
$$S_i = min\{S_j | T_j \in P \cap subtasks(T_i)\}$$
$$E_i = max\{E_j | T_j \in P \cap subtasks(T_i)\}. \quad (7)$$

Notice that the duration of a compound task is defined by the time allocation of its subtasks while the duration of a primitive task is defined by the activity.

A *feasible process* is a process where the time variables S_i and E_i of tasks in the process can be instantiated in such a way that they satisfy the above temporal constraints. It is easy to realize that if there are no additional constraints then any process is feasible. The process defines a partial order of tasks so their start and end times can be set in the left-to-right order while satisfying the constraints (6) and (7).

The nested structure may not be flexible enough to describe naturally some additional relations in real-life processes, for example when an alternative for one task influences the selection of alternatives in other tasks. The following constraints can be added to the nested structure to simplify description of these additional relations between any two tasks T_i and T_j (Barták et al. 2011):

precedence $(i \rightarrow j)$: $T_i, T_j \in P \Rightarrow E_i \leq S_j$ (8)
start-start sync. $(i\ ss\ j)$: $T_i, T_j \in P \Rightarrow S_i = S_j$ (9)
start-end sync. $(i\ se\ j)$: $T_i, T_j \in P \Rightarrow S_i = E_j$ (10)
end-start sync. $(i\ es\ j)$: $T_i, T_j \in P \Rightarrow E_i = S_j$ (11)
end-end sync. $(i\ ee\ j)$: $T_i, T_j \in P \Rightarrow E_i = E_j$ (12)
mutual excl. $(i\ mutex\ j)$: $T_i \notin P \vee T_j \notin P$ (13)
equivalence $(i \Leftrightarrow j)$: $T_i \in P \Leftrightarrow T_j \in P$ (14)
implication $(i \Rightarrow j)$: $T_i \in P \Rightarrow T_j \in P$ (15)

Note that if extra constraints are used then the existence of a feasible process is no longer guaranteed. For example an equivalence constraint between the tasks Buy and $Welding$ in Figure 1 causes no feasible process to exist.

In summary, we can model the nested workflow with extra constraints as a tuple $W = (Parallel, Alternative, Primitive, root, parent, D, \mathcal{C})$, where the $parent$ relation defines a tree rooted at the node $root$ with leaves $Primitive$ and internal nodes $Parallel \cup Alternative$. The set $\mathcal{C}$ defines the extra constraints (8) – (15) and D maps the tasks in $Primitive$ to non-negative integers defining durations of primitive tasks. The process is a subtree of this tree satisfying constraints (3) – (5) and (13) – (15), where each node T_i has assigned two integers S_i and E_i satisfying the constraints (6) – (12).

Robust Free Order Dependency Grammar

The paper (Barták 2016) answers a question whether it would be possible to unify various hierarchical structures of workflows using a single concept. The answer is relatively simple – attribute grammars (Knuth 1968) are suggested as a unifying concept for the description of hierarchical workflows. Figure 2 shows a complete attribute grammar modeling the nested workflow from Figure 1. Each grammar

$Chair(S_{chair}, E_{chair}) \rightarrow$
$Parts(S_{parts}, E_{parts}).$
$\ Assembly(S_{assembly}, E_{assembly})$
$[S_{chair} = min\{S_{parts}, S_{assembly}\},$
$E_{chair} = max\{E_{parts}, E_{assembly}\},$
$E_{parts} \leq S_{assembly},$
$S_{assembly} + D_{assembly} = E_{assembly}]$

$Parts(S_{parts}, E_{parts}) \rightarrow$
$Legs(S_{legs}, E_{legs}).Seat(S_{seat}, E_{seat}).$
$\ Back(S_{back}, E_{back})$
$[S_{parts} = min\{S_{legs}, S_{seat}, S_{back}\},$
$E_{parts} = max\{E_{legs}, E_{seat}, E_{back}\}]$

$Legs(S_{legs}, E_{legs}) \rightarrow$
$Saw1(S_1, E_1), Saw2(S_2, E_2), Saw3(S_3, E_3),$
$\ Saw4(S_4, E_4)$
$[S_{legs} = min\{S_1, S_2, S_3, S_4\},$
$E_{legs} = max\{E_1, E_2, E_3, E_4\},$
$S_1 + D_1 = E_1, S_2 + D_2 = E_2, S_3 + D_3 = E_3,$
$S_4 + D_4 = E_4]$

$Seat(S_{seat}, E_{seat}) \rightarrow$
$Cutting(S_{cut}, E_{cut}).Polishing(S_{polish}, E_{polish})$
$[S_{seat} = min\{S_{cut}, S_{polish}\},$
$E_{seat} = max\{E_{cut}, E_{polish}\},$
$E_{cut} \leq S_{polish}, S_{cut} + D_{cut} = E_{cut},$
$S_{polish} + D_{polish} = E_{polish}]$

$Back(S_{back}, E_{back}) \rightarrow$
$Buy(S_{buy}, E_{buy})$
$[S_{back} = S_{buy}, E_{back} = E_{buy}, S_{buy} + D_{buy} = E_{buy}]$

$Back(S_{back}, E_{back}) \rightarrow$
$Weld(S_{weld}, E_{weld})$
$[S_{back} = S_{weld}, E_{back} = E_{weld},$
$S_{weld} + D_{weld} = E_{weld}]$

Figure 2: An attribute grammar modeling the nested workflow from Figure 1.

symbol has two attributes S and E representing the start and end times of the corresponding task. Symbol D is a constant representing duration of primitive activities. The constraints attached to grammar rules are describing particular relations as specified in (6) – (7).

This theoretical result shows that the transformation of a workflow into an attribute grammar is feasible. Let us now make one step further and to search for a concrete interpreter of an attribute grammar which would serve not only as a theoretical framework for the description of workflows, but it will also make it possible to interpret and verify concrete grammars describing concrete workflows.

In the literature it is often mentioned that attribute grammars are typically used in compiler construction. Another application area for attribute grammars is very often neglected – the attribute grammars are also suitable for natural language parsing, their ability to hold a wide variety of attributes and their values in one grammar symbol enables to hand-craft grammars describing complex phenomena in natural languages. The endeavor to verify workflows by means of attribute grammars actually seems to have certain common features with the process of checking the grammatical correctness of natural language sentences.

In this paper we would like to discover whether the grammar checking of natural languages (and the type of attribute grammar which has been developed for this task) can also be used for the verification of workflows (and which changes or modifications are required). This would not, of course, completely solve the problem of verification of nested workflows, because there are in fact two types of verification. One is the decision whether a given process consisting of primitive tasks fulfills the criteria and constraints of the model of nested workflows or if such a process can be generated by a particular attribute grammar. This is the type of verification where we believe that it actually resembles the process of grammar checking of natural language sentences.

The second type of verification, which we are not going to address in this paper, is the verification of the model itself. This represents the verification whether a given attribute grammar actually generates any process and/or finding the nonterminal(s) which cannot be used in generation of the workflow. This research direction is also very important and interesting, but it goes beyond the scope of this paper.

One implementation of a certain type of an attribute grammar serving primarily for grammar checking of a natural language with a high degree of word order freedom has been completed some time ago for Czech (Holan, Kuboň, and Plátek 1997). In the remaining part of this paper we would like to show that this particular implementation might serve (with minor extensions) also for workflow description.

The implementation of the Robust Free Order Dependency Grammar (RFODG) has been thoroughly described in (Holan 2001). Let us now mention those properties of its implementation which are relevant for our purpose.

Concepts Used in the RFODG

The input data for RFODG had the form of a sequence of data items representing individual word forms and punctuation marks from the input sentence. Each data item had a general form of a list of attributes and their values. Some attributes were obligatory (the attributes describing the original word form, its lemma, Part–of–speech information and syntactic properties), the rest of attributes had been optional. One specific attribute is allowed for complex values in a form of a list of attribute–value pairs. This attribute represented a valency of a word, i.e. the list of obligatory requirements on properties of dependent words. The requirements described in this list had to be fully satisfied before the analysis could go on. The list has been introduced in order to make it possible to verify whether all words required by a particular governing word (a verb in most cases) are actually present in the input sentence. For example, the verb $to\ give$ requires at least three (obligatory) dependent words representing a subject (who), object (what) and indirect object (to whom), thus its list contained three sets of requirements, one for each dependent word.

The grammar rules of RFODG have a common general form $AB \Rightarrow X$ (RFODG had been used as an analytical grammar) where the letters A and B represent two data items from the input sentence and A stands (immediately) to the left from B (i.e., the order of those two items in the input sentence is relevant). In case that some rule is successfully applied to items A and B, a new item X is created (X inherits all attributes and their values from a dominant item – the dominance of either A or B is explicitly expressed by assignments $X := B$ or $X := A$. Each grammar rule is interpreted as a sequence of tests or assignments, it actually expresses a procedural description of the process of checking the applicability of a given rule to a particular pair of items A and B. For handling the list of valency requirements, a special temporary item P is used.

The grammar uses also the following concepts relevant for our purpose:

- Hard (A.x = B.y) and soft (A.x ? B.y ERR) constraints. The concept of soft constraints allows to capture grammatical errors. If a soft constraint is violated, the processing continues, but the rule inserts an error flag ERR into the syntactic tree of the input sentence. If a hard constraint is violated, the processing of the rule immediately stops for the given pair of A and B.
- Branching in a form of a simple IF THEN ELSE ENDIF command.
- The choice of an element from the list (P in A.x). The temporary item P makes it possible to test all members of a list of valency requirements one by one. If, for example, the item A represents an object and the item B a main verb of the input sentence, the temporary item P sequentially represents all items in the valency list of B, until it finds which of those items actually has the requirements consistent with the attributes of A.
- Deleting from the list of valency requirements ($\setminus$ P from X.x). If P successfully identifies a valency requirement consistent with the attributes of A, the corresponding requirement is deleted from the list. This operation guarantees that each requirement is satisfied only once.
- Constants OK and FAIL marking a successful and unsuccessful end of application of a particular grammar rule to a given pair of data items.
- Comments (any text following a semicolon located at the beginning a row of a text)

Let us now present a sample grammar rule in RFODG. It processes the incongruous attribute standing to the right of a governing noun.

```
IF A.SYNTCL = noun THEN ELSE
  IF A.SYNTCL = prephr THEN ELSE
    FAIL
  ENDIF
ENDIF
B.SYNTCL = noun
B.CASE = gen
A.RIGHTGEN ? yes  Second_genitive
X:=A
X.RIGHTGEN := no
OK
END_P
```

The rule works in the following way:

The symbol A must represent either a noun or a prepositional group. If not, the application of the rule immediately fails. The symbol B must represent a noun in the genitive case. The soft constraint checks whether this is the first incongruous attribute in genitive case being attached to the same noun. If not, it inserts an error flag. Then A is selected as the governing word and X inherits all its attributes. The subsequent assignment($X.RIGHTGEN := no$) marks the fact that the incongruous attribute in genitive case is being attached and thus no other such attribute may be attached in the future. The keyword OK confirms a successful application of the rule on the given pair of symbols A and B.

Workflow Verification by means of RFODG

RFODG has been developed as an analytical grammar, therefore the verification of a concrete workflow is the most natural task which we may use it for. Let us start with the basic concepts first.

As it was already mentioned above, each workflow may be decomposed into the primitive tasks by three decomposition operations, namely parallel, serial and alternative decompositions. Let us now look at the possible ways how these three operations may be described in the RFODG.

Serial decomposition

Serial decomposition seems to be the most natural one for RFODG. Two primitive tasks in a sequence actually correspond to two word forms of a natural language, one preceding the other in a sentence. RFODG distinguishes the mutual position of items A and B (A standing always to the left of B), therefore in the rule describing the sequence *Create parts* and *Assembly* from our workflow example, A will represent the task *Create parts* and B will correspond to *Assembly*. The first two commands of a grammar rule describing this sequence may then look like this:

```
A.task = Create_parts
B.task = Assembly
```

Unlike in the analysis of a natural language, in the workflow verification it is not necessary to distinguish between the governing (dominant) and dependent item, therefore the assignment of A or B to X is practically arbitrary. We only must make sure that all relevant attributes are transferred from both tasks to X. For example, if both A and B have attributes marking the start and end time of each task, the rest of our grammar rule might look like this:

```
X := A
X.task := Chair
X.end := B.end
OK
END_P
```

It is not necessary to transfer explicitly the value of A.start because all attributes and their values from A are inherited by X.

Alternative decomposition

Alternative decomposition can also be handled relatively easily. RFODG makes it possible to handle alternatives by means of a conditional statement IF, as we can see in the following example of a grammar rule for the alternative tasks of Buying and Welding.

```
IF A.task = Buy THEN ELSE
  IF A.task = Welding THEN ELSE
    FAIL
  ENDIF
ENDIF
X := A
X.task := Back_support
OK
END_P
```

This grammar rule has two possible outcomes, the resulting item *Back support* will consists either from the task *Buy* or from the task *Welding*, but it cannot include both primitive tasks. If the input sequence of primitive tasks will contain both alternatives, the grammar interpreter will apply this rule twice, once for each primitive task, but because the grammar will not contain a grammar rule combining together *Back support* with any of the two primitive tasks, the grammar will fail and thus it will indicate that the input sequence is incorrect.

Parallel decomposition

Unlike in the theoretical model described in the paper (Barták and Rovenský 2014) where the parallel decomposition served even as a possible model for serial decomposition, it constitutes a serious issue for verification. Just recall that parallel decomposition means that child tasks can be used in any order. A specific order is used in the grammar rule, but this order can be arbitrary and does not correspond to actual temporal allocation of tasks (Barták 2016). On the first sight it seems that verifying primitive tasks which constitutes a parallel decomposition of some more complex tasks is very similar to processing of valency constituents in natural language processing. The obligatory constituents (such as subject, direct and indirect object etc.) are also in a certain sense parallel – they are immediate children of a governing verb, they occupy the same level in a syntactic tree and all of them must be present in a sentence (or they must be identifiable from the previous context if they are not present), otherwise the sentence would not be complete.

Unfortunately, there is one substantial difference which makes this idea invalid. In a natural language sentence, the governing word is actually also present in the input sentence, therefore we can easily identify all constituents from its valency frame and analyze them as direct dependents of the verb. In a nested workflow, the complex task which has been parallely decomposed into several primitive tasks is not present in the input and therefore it may not provide the information about the primitive tasks it was decomposed into.

This actually means that the information which tasks are parallel has to be incorporated into the grammar. Instead of having a single grammar rule for each decomposition, we have to combine n-1 grammar rules together in a way which will guarantee that all primitive parallel tasks are present in the input sequence. For this purpose we might exploit some technical items which whose only role will be to bind the set of grammar rules together. The solution for the complex task *Legs* which has been parallely decomposed into four primitive tasks *Saw1*, *Saw2*, *Saw3*and *Saw4* might then look for example like this (It is of course necessary to make sure that all necessary attributes from *Saw2*, *Saw3* and *Saw4* are properly inherited, this is not contained in the sample grammar rules.):

```
A.task = Saw1
B.task = Saw2
X := A
X.task := Saw12
OK
END_P

A.task = Saw12
B.task = Saw3
X := A
X.task := Saw123
OK
END_P

A.task = Saw123
B.task = Saw4
X := A
X.task := Legs
OK
END_P
```

There is one more issue which might require certain modifications of RFODG. The above mentioned grammar rules would work only in the case that the parallely decomposed primitive tasks are ordered (we suppose that although they are parallel, they will constitute a part of a input sequence of tasks) in the same way as it is expected by the grammar rules, i.e. *Saw1* will stand to the left from *Saw2* and so on. This ordering of the input represents the application of certain additional input constraints, different from the the actual order of individual tasks in time. Let us note that these constraints are not going to contradict the time constraints because the latter ones are going to be explicitly written in the attributes and thus they don't have to be repeated in the input. Possible modification of the RFODG formalism so that it will be able to ignore the general requirement for an item A being to the left from item B for a specified set of grammar rules the constitutes a second option how to solve this issue.

Constraints

If we look at additional constraints which might be used in nested workflows, the case of constraints (1)-(7) from the section Nested Workflows seems to be relatively easy. These simple constraints will be transformed into attributes

and their values. As it was mentioned above, RFODG can deal with two types of constraints, hard and soft ones and thus it has enough expressive power to deal also with these constraints for nested workflows. The only difference is the fact that the constraints used in RFODG are able to compare the values of individual attributes on equality only. This is caused by the expectation that values of attributes always have the form of text, using any other data type does not make sense in natural language analysis. For the smooth transformation of constraints from nested workflows into RFODG constraints it will be necessary to enrich the set of applicable data types and comparison operators. This should be merely a technical issue. As for the constraints of types (8)-(15), their representation in the RFODG will require further investigation.

Conclusions

This paper describes the first step towards bridging the gap between two relatively distant areas – the nested workflows and grammar checking of natural languages. Concrete types of grammar rules in the formalism originally developed for grammar checking (RFODG) have been suggested for all three main types of decomposition of nested workflows. It turned out that the similarity of the two tasks is not absolute, that one of those types, a parallel decomposition, might require a modification of the RFODG formalism or the exploitation of specific constraints governing the order (left to right) of primitive tasks in the input. The natural next step in our research will be the investigation of additional, more complex constraints and their transformation into RFDG. Last but not least is then the research whether RFODG may also be used as a generative grammar, whether it will be possible (maybe again after a slight modification) to generate a sequence of primitive tasks by means of this type of an attribute grammar.

Acknowledgments

The research reported in this paper has been supported by the Czech Science Foundation GACR, grant No. P103-15-19877S

References

Bae, J.; Bae, H.; Kang, S.-H.; and Kim, Z. 2004. Automatic control of workflow processes using eca rules. *IEEE Transactions on Knowledge and Data Engineering* 16:1010–1023.

Barták, R., and Rovenský, V. 2014. On verification of nested workflows with extra constraints: From theory to practice. *Expert Systems with Applications* 41:904–918.

Barták, R., and Čepek, O. 2008. Nested Temporal Networks with Alternatives: Recognition, Tractability, and Models. In *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA 2008)*, 235–246. Springer Verlag: LNAI.

Barták, R.; Cully, M.; Jaška, M.; Novák, L.; Rovenský, V.; Sheahan, C.; and Skalický, T. 2011. Workflow Optimization with FlowOpt, On Modelling, Optimizing, Visualizing, and Analysing Production Workflows. In *Proceedings of Conference on Technologies and Applications of Artificial Intelligence (TAAI 2011)*, 167–172. IEEE Conference Publishing Services.

Barták, R. 2016. Using Attribute Grammars to Model Nested Workflows with Extra Constraints. In *SOFSEM 2016: Theory and Practice of Computer Science*, 171–182. Springer Verlag.

Holan, T.; Kuboň, V.; and Plátek, M. 1997. A Prototype of a Grammar Checker for Czech. In *Proceedings of the Firth Conference on Applied Natural Language Processing*. Washington, DC: ACL.

Holan, T. 2001. *Nástroje pro vývoj závislostních anayzátorů přirozených jazyků s volným slovosledem.* Prague: MFF UK. (in Czech).

Knuth, D. E. 1968. Semantics of context-free languages. *Mathematical Systems Theory* 2:127–145.

van der Aalst, W., and t. Hofstede, A. H. M. 2005. Yawl: yet another workflow language. *Information Systems* 30:245–275.