Simplifying Verification of Nested Workflows with Extra Constraints

Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics, Malostranské nám. 25, 118 00 Praha 1, Czech Republic
bartak@ktiml.mff.cuni.cz

Abstract

Workflow verification is an important aspect of workflow modeling where the verification task is to ensure that the workflow describes feasible processes. It has been shown that verifying nested workflows with extra precedence, causal, and temporal synchronization constraints is an NP-complete problem and a verification method based on constraint satisfaction has been proposed. This paper theoretically justifies the task-collapsing component of this method and provides examples of easy-to-verify constraints.

Introduction

The importance of process description has increased in the enterprise information systems in recent 15 years. At the same time, various case studies showed that process designers tend to make many errors when describing the process (van der Aalst et al. 2011). Hence, the methods to formally verify the process models are becoming increasingly important.

A *workflow* is a formal description of a set of related processes. It consists of activities that can be grouped to tasks and that are connected via constraints such as temporal relations. In this paper we focus on a particular type of workflows called *nested workflows* that are obtained by decomposition operations. To increase flexibility of these workflows for real-life problems they can be extended by extra precedence, causal, and temporal synchronization constraints between the tasks (Barták et al. 2011). The *verification problem* consists of determining if each task can appear in some feasible process. If this is the case, the workflow is called *sound*. The nested workflows are always sound but if we add the extra constraints then the verification problem becomes intractable (Barták, 2012). A constraint-based method to verify nested workflows was proposed in (Barták and Rovenský 2013). This method uses an efficiency enhancement based on collapsing some tasks. In this paper we will formally define the task collapsing method and we will show under which conditions the task collapsing can be used in the verification process. We will also present several examples of extra constraints that can be easily verified (are either violated or redundant).

The paper is organized as follows. We will first give a formal definition of the nested workflow with extra constraints. Then we will formulate the verification problem and sketch the verification method from (Barták and Rovenský 2013). After that we will justify the method of task collapsing. Finally, we will describe the method to identify some constraints as redundant or violated.

Nested Workflows with Extra Constraints

There exist many formal models of workflows such as BPMN (Business Process Modeling Notation, www.bpmn.org) or YAWL (Yet Another Workflow Language, www.yawlfoundation.org). These are very generic models that support repetition of parts of the workflow and many specific constraints between the activities. In this work we use nested workflows from the FlowOpt system (Barták et al. 2011) that are based on the formal model of a Nested Temporal Network with Alternatives (Barták and Čepek 2008). The nested structure resembles the idea of hierarchical task networks leading to Temporal Planning Networks (Kim et al. 2001) and it is a quite common structure in real-life workflows (Bae et al. 2004).

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 decomposition. Figure 1 gives an example of a nested workflow that shows how the tasks are decomposed. The root task *Chair* is decomposed serially into 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 an example of alternative decomposition into two primitive tasks with *Buy* and *Welding* activities (*Welding* is treated as an alternative to *Buy*). Naturally, the nested workflow can also be described as a tree of tasks (Figures 1 and 2). Formally, the nested workflow is a set *Tasks* of tasks that is a union of four disjoint sets: *Parallel*, *Alternative*, *Serial*, 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 ∈ *Tasks* | *parent*(C) = T }). The tasks from the sets *Parallel*, *Alternative*, and *Serial* are called *compound tasks* and they decompose to some subtasks:

$$T \in (Parallel \cup Alternative \cup Serial) \Rightarrow subtasks(T) \neq \emptyset,$$

while the primitive tasks do not decompose:

$$T \in Primitive \Rightarrow subtasks(T) = \emptyset.$$

We can now define *tasks*(T) as the set of all tasks in the decomposition of T:

$$tasks(T) = \cup_{T' \in subtasks(T)} tasks(T') \cup \{T\}.$$

In particular, if T is a primitive task then *tasks*(T) = {T} (because T has not subtasks) and if T is a compound task then *tasks*(T) contains the task T, all subtasks of T, their subtasks etc. For the workflow from Figure 1, *tasks*(Seat) = {Seat, Cutting, Polishing}.

The workflow defines one or more processes. A *process* P is a subset of tasks satisfying the following properties:

- for each task T in the process, that is not a root task, its parent task is also in the process,
  $T \in P \wedge T \neq root \Rightarrow parent(T) \in P,$ (1)
- for each compound task T in the process with a serial or parallel decomposition, all its subtasks are also in the process,
  $T \in P \cap (Serial \cup Parallel) \Rightarrow subtasks(T) \subseteq P,$ (2)
- for each compound task T in the process with the alternative decomposition, exactly one of its subtasks is in the process,
  $T \in P \cap Alternative \Rightarrow |\, subtasks(T) \cap P \,| = 1.$ (3)

So far we defined the hierarchical structure of the nested workflow, but as Figure 1 shows the nested structure also defines certain temporal relations. These temporal relations must hold for all tasks in a single process. Assume that S_T is the start time and E_T is the end time of task *T*. The primitive tasks *T* are filled with activities and each activity has some known positive duration D_T. Then for the tasks in a certain process P the following relations hold:

$$T \in P \cap Primitive \Rightarrow S_T + D_T = E_T \quad (4)$$

$$T \in P \cap (Parallel \cup Alternative \cup Serial) \Rightarrow \quad (5)$$
$$S_T = \min\{ S_C \mid C \in P \cap subtasks(T) \}$$
$$E_T = \max\{ E_C \mid C \in P \cap subtasks(T) \}.$$

Notice that the duration of a compound task is determined by the time allocation of its subtasks while the duration of a primitive task is determined by the activity. Moreover, for the serial decomposition we expect the subtasks to be ordered, say $T_1, \ldots, T_n$, where *n* is the number of subtasks of a given task. Hence, the following constraint must hold for a serial task in the process:

$$T \in P \cap Serial \wedge subtasks(T) = \{ T_1, \ldots, T_n \} \Rightarrow \quad (6)$$
$$\forall i = 1,\ldots, n\text{-}1: E_i \leq S_{i+1}$$

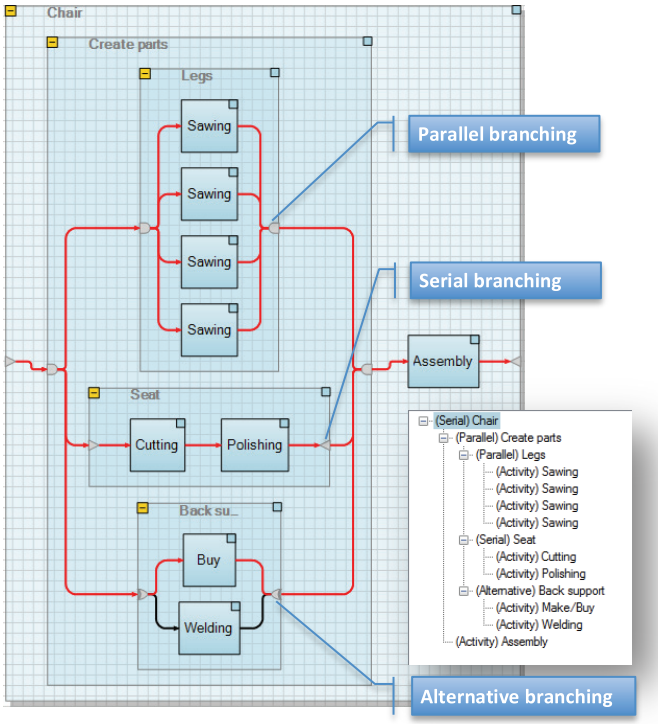

Figure 1. Nested workflow as it is visualized in the FlowOpt Workflow Editor (from top to down there are parallel, serial, and alternative decompositions).

A *feasible process* is a process where the time variables S_T and E_T can be instantiated in such a way that they satisfy the temporal constraints (4)-(6). If there are no extra constraints (see below) 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 all the temporal constraints.

To increase flexibility of nested workflows when describing real-life processes, for example the alternative for one task influences the selection of alternatives in other tasks, the following extra constraints between any two tasks *i* and *j* can be added to the nested structure:

- precedence constraint ($i \rightarrow j$): $i, j \in P \Rightarrow E_i \leq S_j$
- start-start synchronization (i ss j): $i, j \in P \Rightarrow S_i = S_j$
- start-end synchronization (i se j): $i, j \in P \Rightarrow S_i = E_j$
- end-start synchronization (i es j): $i, j \in P \Rightarrow E_i = S_j$
- end-end synchronization (i ee j): $i, j \in P \Rightarrow E_i = E_j$
- mutex constraint (i mutex j): $i \notin P \vee j \notin P$
- equivalence constraint ($i \Leftrightarrow j$): $i \in P \Leftrightarrow j \in P$
- implication constraint ($i \Rightarrow j$): $i \in P \Rightarrow j \in P$.

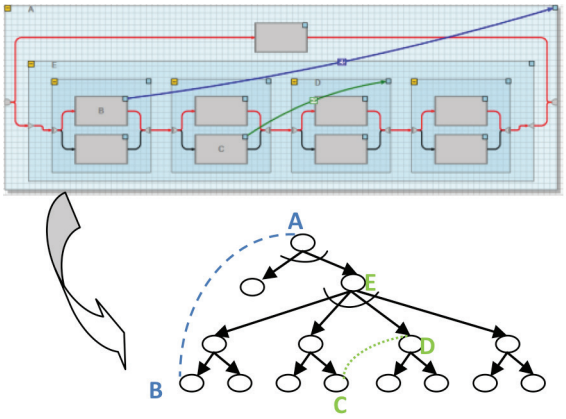

Figure 2. A tree representation of nested workflows with examples of possible locations of extra binary constraints.

There are basically two different locations where the extra binary constraint can be placed in the tree structure of the workflow. Either the constraint connects two tasks on the same path to the root task (for example the constraint between tasks A and B in Figure 2) or the constraint connects the tasks from different sub-trees with a common ancestor task (for example the constraint between tasks C and D in Figure 2). We call this common ancestor task a *parent of the constraint*. The parent of the constraint between A and B in Figure 2 is A, while the parent of the constraint between C and D is E. Formally, if the constraint C connects two tasks *i* and *j* then the *parent*(C) is a task T such that $\{i,j\} \subseteq tasks(T)$ and there is no task T' $\in tasks(T)$ such that $\{i,j\} \subseteq tasks(T')$. As we shall show later, the location of the constraint and its type can be used to deduce whether the constraint is redundant or violated.

It is interesting that if extra constraints are used then the existence of a feasible process is no longer obvious because of the violated constraints (for example, the extra precedence constraints may introduce a loop). In fact, Barták (2012) showed that the problem, whether a feasible process exists or not, is NP-complete. Because the existence of a feasible process is not guaranteed, it is important to verify the workflow structure before the workflow is used further (Giro 2007).

Workflow Verification

Various workflow verification methods have been proposed. The methods based on Petri Nets (van der Aalst and Hofstede, 2000) are probably the most widely applied, but they cannot cover the extra constraints. The verification method for the nested workflows with extra constraints was proposed in (Barták and Rovenský 2013). This method formulates the problem of checking existence of a feasible process as a constraint satisfaction problem (CSP). Basically the set P of tasks in the process is described by Boolean variables indicating which tasks are included in the process and the constraints defining a feasible process (see the previous section) are reformulated using these variables. The verification algorithm repeatedly solves CSPs where each CSP describes a feasible process containing a given task (the Boolean variable for that task is set to *true*). If all CSPs have solutions then the workflow is sound, otherwise a flawed task is identified.

Task Collapsing

Barták and Rovenský (2013) showed experimentally that collapsing some tasks and treating them as primitive tasks in the verification algorithm speeds up the core verification method. In this paper we will formalize the process of task collapsing and prove that the verification method based on task collapsing is sound.

It is easy to observe that if we take the set *tasks*(T) and all the constraints from the workflow W over the tasks from *tasks*(T) then we obtain a nested workflow with the root task T. Let us denote this workflow W_T. Conversely, we can cut this workflow out of W and leave there only the task T (instead of W_T), which is called *task collapsing*. Formally, workflow cW_T obtained from the workflow W by collapsing its task T is defined by the set of tasks $tasks(cW_T) = (tasks(root) \setminus tasks(T)) \cup \{T\}$, where *root* is the root task of W. The task T has no subtasks in cW_T so it is a primitive task there. We set its duration $D_T = 1$ (any positive number can be used). All the constraints from W over the tasks from $tasks(cW_T)$ are preserved in cW_T. Figure 3 shows how the workflow splits to workflows W_T and cW_T (a tree representation of the workflow is used).

The motivation behind the task collapsing is that under certain conditions we can verify the workflow W by verifying the smaller workflows W_T and cW_T. Notice that all the workflow constraints (1)-(6) from W are either in W_T or in cW_T. It may only happen that an extra constraint is defined between a task from *tasks*(T) and a task outside *tasks*(T) and hence this extra constraint is present neither in

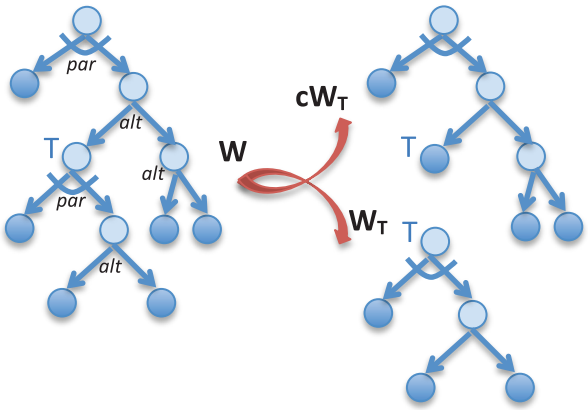

Figure 3. Decomposition of the workflow to the workflow defined by the task T and the workflow obtained by collapsing the task T.

W_T nor in cW_T. Even if there are no such constraints between the workflows, it may happen that extra constraints in cW_T restrict the duration of T that further influences the feasible processes that can be selected from W_T. Assume the situation depicted in Figure 4, where the task T must have identical duration to the primitive task A3 due to the start-start and end-end temporal synchronization constraints between these tasks. Obviously these constraints restrict which subtask A1 or A2 can be selected from the alternative decomposition of T (that one with the duration equal to A3). Hence, we should ensure that the duration of T is not restricted in cW_T (recall that we use $D_T = 1$ in cW_T, but the real duration of T may be different depending on the process selected for W_T). In summary, to ensure that we can independently select feasible processes for workflows W_T and cW_T and combine them to obtain a feasible process for W, there cannot be any extra constraint between the tasks of these processes and the duration of task T cannot be restricted by other tasks in cW_T (only the extra temporal constraints can restrict the task duration).

Proposition 1: Let T be a task of the workflow W s.t.

(a) there is no constraint in W between any pair of tasks T_1 and T_2 such that $T_1 \in tasks(T) \setminus \{T\}$ and $T_2 \in tasks(W) \setminus tasks(T)$, and

(b) if there is a temporal synchronization or a precedence constraint C then $T \notin tasks(parent(C))$.

Then for any feasible process P1 from cW_T and for any non-empty feasible process P2 from W_T the following claims hold:

- if $T \notin P1$ then P1 is a feasible process in W,
- if $T \in P1$ then $P1 \cup P2$ is a feasible process in W.

Proof: Thanks to the condition (a) any constraint from W is either in cW_T or in W_T. This is because any constraint from W connects the tasks that are either in cW_T or in W_T.

Let us assume first that $T \notin P1$. Then P1 is a feasible process in W because it satisfies all the constraints W. The constraints from cW_T are satisfied, because P1 is a feasible process in cW_T. As the task T is not included in P1 then no task from *tasks*(T) is included in this process so all the constraints from W_T are trivially satisfied (an empty process is always feasible).

Let us now assume the remaining case, that is $T \in P1$. $P = P1 \cup P2$ satisfies all the workflow structural constraints (1)-(3) defining the valid decompositions of compound tasks so P is a process in W. In particular, notice that $T \in P2$ because P2 is a non-empty process (a non-empty process always contains the root task of the workflow). The process P also satisfies all the extra logical constraints (mutex, implication, equivalence). These constraints are defined either between the tasks of cW_T or between the tasks of W_T and they allow the task T to be included in P1 and P2 and hence also in P. It remains to show that the temporal constraints (4)-(6), precedences,

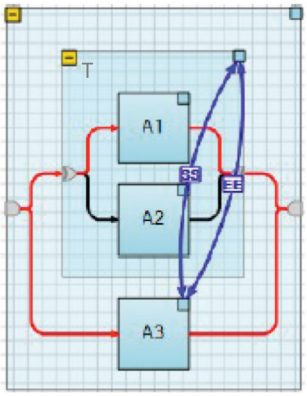

Figure 4. Example of restricting the duration of task via extra synchronization start-start and end-end constraints.

and synchronizations are satisfied. We know that these constraints are satisfied in P2 and they define the duration T. However, in P1 we assumed the duration of T to be 1 that may be different from the duration of T defined by P2. We shall show that the temporal constraints in P1 are satisfied independently of the particular duration of T (there are no constraints connecting tasks in P1 and P2). In particular we will focus on the path from T to the root task in cW_T and we will show that it is possible to modify the durations of tasks on this path while keeping all the temporal constraints satisfied. From the condition (b) we know that no task TP on the path from T to the root task participates in a temporal synchronization or a precedence constraint (otherwise *parent*(C) would also be on that path and then $T \in tasks(parent(C))$). Hence the duration of TP is computed from the durations of its subtasks using the constraint (5). Moreover, the only temporal relation between the subtasks of TP is (6), in case of a serial decomposition – the condition (b) forbids any other temporal relation, direct or indirect, between the subtasks of TP. It means that the subtasks of TP are completely temporally independent (they do not share any start time or end time variables). If we change the duration of any subtask of TP then only the duration of the parent task TP is influenced to satisfy the constraint (5). The consequence is that if we satisfy all the temporal constraints in subtasks independently then by propagating the temporal information through constraints (5)-(6) we satisfy all the temporal constraints in the workflow rooted in TP. Hence for any duration of T, we can compute the durations of tasks in the path from T to the root of W in such way that all the constraints (5)-(6) are satisfied and all other temporal constraints in cW_T remain satisfied. Hence the process P1 where the duration of T is modified to be equal the duration of T in P2 is a feasible process and so P1 and P2 can be combined together and $P1 \cup P2$ is a feasible process in W (it satisfies all the constraints in W). In summary, we showed that a feasible process for W can be composed from the feasible processes for cW_T and W_T. ■

Corollary: If a task T from the workflow W satisfies the conditions of Proposition 1 then the workflow W is sound if and only if the workflows cW_T and W_T are sound.

Proof: Obviously any feasible process P from W can be split to feasible processes P1 for cW_T and P2 for W_T,where $P = P1 \cup P2$, $P1 \subseteq tasks(cW_T)$, and $P2 \subseteq tasks(W_T)$. Let $T' \in tasks(cW_T)$. There is a feasible process P for W such that $T' \in P$ and also a process P1 containing T' for cW_T. Hence cW_T is sound. Let $T' \in tasks(W_T)$. There is a feasible process P for W such that $T' \in P$ and also a process P2 containing T' for W_T. Hence W_T is sound.

There exists a feasible process P1 containing T in cW_T and a feasible process P2 containing T in W_T (both cW_T and W_T are sound). According to Proposition 1 the process $P = P1 \cup P2$ is a feasible process in W containing T.

Let $T' \neq T$ be a task in W such that $T' \in tasks(cW_T)$. There exists a feasible process P1 for cW_T containing T' (cW_T is sound). If $T \in P1$ then we find a feasible process P2 in W_T containing T (it must exist because W_T is sound), otherwise $P2 = \{\}$. According to Proposition 1 the process $P = P1 \cup P2$ is a feasible process in W containing T'.

Let $T' \neq T$ be a task in W such that $T' \in tasks(W_T)$. There exists a feasible process P2 for W_T containing T' (W_T is sound). There also exists a feasible process P1 for cW_T containing T. According to Proposition 1 the process $P = P1 \cup P2$ is a feasible process in W containing T'.

We have shown that if cW_T and W_T are sound then for any task T' there exists a feasible process in W containing T'. Hence W is sound. ■

The Proposition 1 gives a method how to simplify workflow verification. The workflow W can be verified by verifying smaller workflows cW_T and W_T independently. The proposition also gives a method how to split W to workflows cW_T and W_T. It is enough to find a task T satisfying the conditions (a) and (b) from the proposition. This proposition provides a theoretical justification of the task collapsing method from (Barták and Rovenský 2013) where no extra constraints in W_T were allowed. Obviously, if no extra constraints are present in W_T then W_T is sound. Because the presented proposition allows extra constraints in W_T, it generalizes the original collapsing method.

Easily Verifiable Constraints

While the general verification of nested workflows with extra constraints is hard, it is possible to identify certain constraints that are easy to verify. In particular, based on the location and the type of the constraint it is possible to show that certain constraints are always satisfied and hence redundant and that certain constraints forbid inclusion of some tasks in any process and hence make the workflow flawed. Identifying such constraints is practically important as they can be reported to the user immediately with complete information why the constraint is problematic. These constraints can be omitted when the user attempts to add them. The benefit of omitting such constraints is that it may allow collapsing more tasks based on the Proposition 1.

Let us start with the constraints along the path to the root (the parent of the constraint is one of the constrained tasks – Figure 5). If this constraint is a precedence constraint or start-end synchronization or end-start synchronization then the constraint is invalid because it is in conflict with the constraints (5) – the parent task cannot start at or after the child task finishes and the child task cannot start at or after the parent task finishes. Hence the parent of the constraint cannot be used in any process. The mutual exclusion constraint is also invalid because it is in conflict with constraints (1) – the child task requires the parent task to be a part of the same process so the child task can never be used in any process if the mutex constraint is present. If there is no alternative decomposition on the path between the tasks then the equivalence constraint is redundant – constraints (1) and (2) ensure the equivalence of tasks. Assume now that there is an alternative decomposition of some task T in the path between the tasks A and B connected via an equivalence constraint. Because A and B must always be in the same process, the equivalence constraint causes the workflow to be flawed – only one alternative in the task T can be used due to the constraint (3). Similarly, the implication constraint in the direction to the parent task is redundant due to constraints (1). The implication constraint in the opposite direction (starting from the parent task) is invalid, if there is a task with alternative decomposition on the path between the tasks, otherwise the constraint is redundant. The reason is identical to the equivalence constraint. The status of the start-start and end-end synchronizations can be derived only if all the decompositions between A and B are serial. If B is in the first tasks of all the decompositions then the start-start constraint is redundant, otherwise it is invalid due to the constraints (5)-(6). Similarly, if B is in the last tasks of all the decompositions then the end-end constraint is redundant, otherwise it is invalid due to the constraints (5)-(6). Table 1 summarizes the status of all possible extra constraints along the path to the root task.

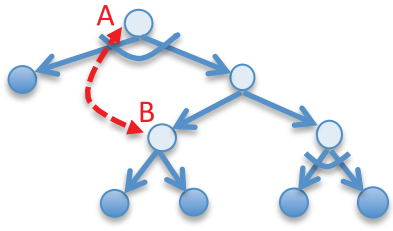

Figure 5. A typical location of an easy-to-verify constraint.

constraint	status	condition
A se B (B es A)	invalid	-
A es B (B se A)	invalid	-
A ss B	redundant	only serial decompositions between A and B, B is always first
	invalid	only serial decompositions between A and B, B not always first
A ee B	redundant	only serial decompositions between A and B, B is always last
	invalid	only serial decompositions between A and B, B not always last
A → B	invalid	-
A ← B	invalid	-
A mutex B	invalid	-
A ⇔ B	invalid	some alternative decomposition between A and B
	redundant	no alternative decomposition between A and B
A ⇒ B	invalid	some alternative decomposition between A and B
	redundant	no alternative decomposition between A and B
A ⇐ B	redundant	-

Table 1: The statuses of constraints between tasks A and B, where A is an ancestor of B (see Figure 5).

Assume now the second case when the parent task of the constraint is different from both constrained tasks. For these constraints we can derive their status only in some very specific situations. If there is no task with alternative decomposition on the path between the tasks then the equivalence and implication constraints are redundant while the mutual exclusion constraint is inconsistent due to the constraints (1) and (2). If the parent task of the constraint is a task with an alternative decomposition then the synchronization and precedence constraints are redundant (the constrained tasks will never be together in the same process due to the constraint (3)). If the parent task is a task with the serial decomposition then a precedence constraint is redundant if it follows the direction of the serial decomposition – due to the constraint (6). It is possible to identify other special cases where the validity of the constraint can be immediately checked, but the conditions become more complex. The key message of this section is that there are certain extra constraints that can be verified during pre-processing of the workflow and immediately and precisely reported to the user.

Conclusions

This paper deals with the theoretical foundations of simplifying verification of nested workflows with extra constraints. It formally defines the method of task collapsing and gives the condition when the workflow can be split to independently verifiable workflows. This theoretically justifies and also generalizes the method from (Barták and Rovenský 2013) that requires the workflow W_T to be without extra constraints. From the formal proof it seems that the condition for task collapsing can be weakened further, which is a topic of future research.

As the second contribution, the paper gives a list of extra constraints that can be easily verified by fast ad-hoc methods. While the general constraint-based verification method from (Barták and Rovenský 2013) can only report a flawed task; these easy-to-verify constraints can be directly reported to the user as the source of problems (if they are invalid). The future research may look at other general methods for finding reasons for flaws, for example similar to identifying nogoods.

Acknowledgements: The research is supported by the Czech Science Foundation under the contract P202/10/1188.

References

Bae, J, Bae, H., Kang, S.-H., Kim, Z., 2004. Automatic Control of Workflow Processes Using ECA Rules. *IEEE Transactions on Knowledge and Data Engineering*, **16**(8), 1010-1023.

Barták, R., and Čepek, O. 2008. Nested Temporal Networks with Alternatives: Recognition, Tractability, and Models. In *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA 2008),* LNAI 5253, Springer Verlag, pp. 235-246.

Barták, R., Cully, M., Jaška, M., Novák, L., Rovenský, V., Sheahan, C., Skalický, T., Thanh-Tung, D. 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)*, IEEE Conference Publishing Services, pp. 167-172.

Barták, R. 2012. On Complexity of Verifying Nested Workflows with Extra Constraints, *Proceedings of 4th International Conference on Agents and Artificial Intelligence (ICAART 2012),* Volume 1, pp. 346-354, SciTePress.

Barták, R., and Rovenský, V. 2013. Verifying Nested Workflows with Extra Constraints. In *MICAI 2012, Part I*, LNAI 7629, p. 359-370, Springer Verlag.

Giro, S. 2007. Workflow Verification: A New Tower of Babel. In *AIS-CMS International Modeling and Simulation Multiconference*, Buenos Aires, Argentina.

Kim, P., Williams, B., Abramson, M. 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *Proceedings of International Joint Conference on Artificial Intelligence* (IJCAI), pp. 487-493, (2001).

van der Aalst, W. M. P., and ter Hofstede, A. H. M. 2000. Verification of Workflow Task Structures: A Petri-Net-Based Approach. *Information Systems*, **25**(1), 43-69, (2000).

van der Aalst, W. M. P., vanHee, K. M., ter Hofstede, A. H. M., Sidorova, N., Verbeek, H. M. W., Voorhoeve, M., and Wynn, M. T. 2011. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23: 333–363.