Automated Resource Allocation in Business Processes with Answer Set Programming*

Giray Havur, Cristina Cabanillas, Jan Mendling, and Axel Polleres

Vienna University of Economics and Business, Austria
{firstname.lastname}@wu.ac.at

Abstract. Human resources are of central importance for executing and supervising business processes. An optimal resource allocation can dramatically improve undesirable consequences of resource shortages. However, existing approaches for resource allocation have some limitations, e.g., they do not consider concurrent process instances or loops in business processes, which may greatly alter resource requirements. This paper introduces a novel approach for automatically allocating resources to process activities in a time optimal way that is designed to tackle the aforementioned shortcomings. We achieve this by representing the resource allocation problem in Answer Set Programming (ASP), which allows us to model the problem in an extensible, modular, and thus maintainable way, and which is supported by various efficient solvers.

Keywords: Answer Set Programming, business process management, resource allocation, timed Petri net, work scheduling

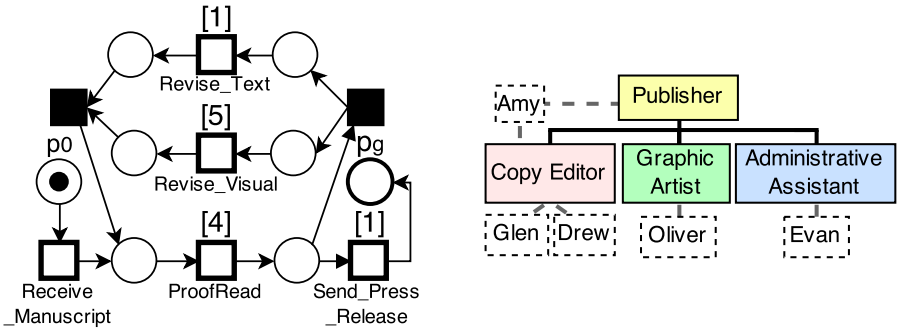
1 Introduction

Human resources[1] are crucial in business process management (BPM) as they are responsible for process execution or supervision. A lack of resources or a suboptimal work schedule may produce delayed work, potentially leading to a reduced quality and higher costs.

In this paper, we address the problem of allocating the resources available in a company to the activities in the running process instances in a time optimal way, i.e., such that process instances are completed in the minimum amount of time. Our approach lifts limitations of prior research pursuing similar goals, which assumes simplified non-cyclic processes and does not necessarily search for an optimal resource allocation [1,2]. To this end, we rely on Answer Set Programming (ASP) [3], a declarative knowledge representation and reasoning formalism that is supported by a wide range of efficient solvers. ASP has been successfully used to address planning and configuration problems in other domains [4].

Our solution is divided into three layers: The core layer represents process models in ASP. The second layer adds all the information related to time, such

* Funded by the Austrian Research Promotion Agency (FFG), grant 845638 (SHAPE).
[1] From now on *resources* for the sake of brevity.



(a) Process to publish a book (b) Organisational model

Fig. 1: Running example

as the estimated activity durations. Finally, resource-related information including, among others, the characteristics of the resources available according to an organisational model as well as the conditions that must be fulfilled to assign resources to activities (e.g., to have a specific organisational role), is encoded on top of these two layers. An ASP solver can use all this data to compute possible optimal solutions for the resource allocation problem. We have evaluated our approach with a proof-of-concept implementation and we have measured its performance with non-trivial scenarios that contain loops and concurrent process instances.[2]

Our modular encodings in ASP provide flexibility and extensibility so that, e.g., additional instances of pre-defined processes can be added. In addition, the declarative nature of the encodings of constraints enables an executable specification of the problem.

The paper is structured as follows: Section 2 presents a scenario that motivates this work as well as related work. Section 3 defines technical background required to understand our approach. Section 4 describes our modular approach for resource allocation in business processes with ASP. Section 5 presents the evaluations performed and Section 6 concludes and outlines future work.

2 Background

In the following, we describe an example scenario that motivates this work and shows the problems to be addressed, and then we outline related work.

2.1 Running Example

In this paper we rely on (timed) Petri nets [5] for business process modelling, commonly used for this purpose due to their well defined semantics and their analysis capabilities. Nonetheless, any process modelling notation can be used with our approach as long as it can be mapped to Petri nets, for which several transformations have already been defined [6]. Fig. 1a depicts a model representing the process of publishing a book from the point of view of a publishing entity. In particular, when the publishing entity receives a new textbook manuscript from an author, it must be proofread. If changes are required, the

[2] Our encoding and the problem instances are provided at `http://goo.gl/lzf1St`

modifications suggested must be applied on text and figures, which can be done in parallel. This review-and-improvement procedure is repeated until there are no more changes to apply, and the improved manuscript is then sent back to the author for double-checking. In Fig. 1a, the numbers above the activities indicate their (default maximum) duration in generic time units (TU)[3].

The organisational model depicted in Fig. 1b shows the hierarchy of roles of a publishing entity. Specifically, it has four roles and five resources assigned to them. The following relation specifies how long it takes to each role and resource to complete the process activities: $(Role \cup Resource) \times Activity \times TU \supset$ {(Copy Editor, Proofread, 2), (Glen, Proofread, 5), (Drew, Proofread, 2), (Drew, Revise Text, 2)}. For resource allocation purposes, the duration associated with a specific resource is used in first place followed by the duration associated with roles and finally, the duration of activities (cf. Fig. 1a). Resources are assigned to activities according to their roles. In particular, the relation activity-role in this case is as follows: $Role \times Activity \supset$ {(Publisher, Receive Manuscript), (Copy Editor, Proofread), (Copy Editor, Revise Text), (Graphic Artist, Revise Visual), (Admin. Asst., Send Press Release)}.

For the purpose of planning the allocation of resources to process activities in an optimal way, the following aspects must be taken into consideration: (i) several process instances can be running at the same time; (ii) the review-and-improvement procedure is a loop and hence, it may be repeated several times in a single process instance. Since one cannot know beforehand the number of repetitions that will be required for each process instance, assumptions must be made about it. Optimality is reached when the activities in all instances of a business process are assigned resources so that the overall execution of all instances takes as little time as possible.

2.2 Related Work

The existing work on resource allocation in business processes has mostly relied on Petri nets. In fact, the goal we pursue is doable at the Petri net level with some shortcomings and limitations. Van der Aalst [1] introduced a Petri net based scheduling approach to show that the Petri net formalism can be used to model activities, resources and temporal constraints with non-cyclic processes. However, modelling this information for multiple process instances leads to very large Petri nets. Moreover, other algorithms for resource allocation proved to perform better than that approach [7]. Rozinat et al. [2] used Coloured Petri nets (CPNs) to overcome the problems encountered in traditional Petri nets. In CPNs, classes and guards can be specified to define any kind of constraints. However, the approach proposed is greedy such that resources are allocated to activities as soon as they are available, overlooking the goal of finding an optimal solution. This may make the allocation problem unsatisfiable.

Several attempts have also been done to implement the problem as a constraint satisfaction problem. For instance, Senkul and Toroslu [8] developed an architecture to specify resource allocation constraints and a Constraint Program-

[3] Please, note that events are instantaneous, and hence, they take zero time units.

ming (CP) approach to schedule a workflow according to the constraints defined for the tasks. However, they aimed at obtaining a feasible rather than an optimal solution and the approach does not support the schedule of concurrent workflows. Besides, Heinz and Beck [9] demonstrated that models such as Constraint Integer Programming (CIP) outperform the standard CP formulations. In addition, loops are disregarded in these approaches.

Resource allocation in projects has been widely investigated [10, 11]. However, projects differ from business processes in that they are typically defined to be executed only once and decision points are missing. Therefore, the problem is approached in a different way. The agent community has also studied how to distribute a number of resources among multiple agents [12,13]. Further research in necessary to adapt those results to resource allocation in business processes [14].

3 Preliminaries

Timed Petri Nets [15] associate durations with transitions: a *timed Petri net* is a 5-tuple $N_T = \langle P, T, F, c, M_0 \rangle$ such that P is a finite set of *places*, T is a finite set of *transitions*, with $P \cap T = \emptyset$, $F \subset (P \times T) \cup (T \times P)$ describes a bipartite graph, M_0 is the *initial marking*. and $c : T \to \mathbb{N}$ is a function that assigns firing delays to every transition $t \in T$. Here, a *marking*(state) $M : P \to \mathbb{Z}^+$ assigns to each place a non-negative integer, denoting number of tokens in places. For each $t \in T$ the *input place set* $\bullet t = \{p \in P \mid (p, t) \in F\}$. The output place set $t\bullet$, and analogously input $\bullet p$ (and output $p\bullet$,resp.) transition sets of a place $p \in P$ can be defined analogously. A transition may *fire*, written $\xrightarrow{t}$, when all $p \in \bullet t$ have tokens: all tokens in $\bullet t$ are consumed and tokens produced in each $p \in t\bullet$.

A Fig. 1a shows an example of a timed Petri net: circles represent places, squares represent transitions, and numbers in brackets on transitions denote firing delays. Filled squares denote "silent" transitions that have no firing delays, i.e., $c(t) = 0$. However, note that also normal transitions that correspond to activities can have no delay, e.g., t_m in Fig. 1a.

A marking M_k is *reachable* from M_{k-1} in *one step* if $M_{k-1} \xrightarrow{t_{k-1}} M_k$. A firing sequence of transitions $\overrightarrow{\sigma} = \langle t_1 t_2 ... t_n \rangle$ changes the state of the Petri net at each firing: $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{\cdots} M_n$. In this paper we use *1-safe Petri nets*, i.e., each place contains at most one token in any state. N_T is called *sound* if from every reachable state, a proper final state can be reached in N. N_T is called *free-choice* if every for transitions t_1 and t_2, $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ implies $\bullet t_1 = \bullet t_2$.

Answer Set Programming (ASP) [3, 16] is a declarative (logic-programming-style) paradigm for solving combinatorial search problems

An *ASP program* Π is a finite set of rules of the form

$$A_0 :\!\text{-} A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n. \tag{1}$$

where $n \geq m \geq 0$ and each $A_i \in \sigma$ are (function-free first-order) atoms; if A_0 is empty in a rule r, we call r a constraint, and if $n = m = 0$ we call r a fact. Whenever A_i is a first-order predicate with variables within a rule of the form (1), this rule is considered as a shortcut for its "grounding" $\mathit{ground}(r)$, i.e., the set of its ground instantiations obtained by replacing the variables with all

possible constants occurring in Π. Likewise, we denote by $ground(\Pi)$ the set of rules obtained from grounding all rules in Π.

Sets of rules are evaluated in ASP under the so-called stable-model semantics, which allows several models (so called "answer sets"), that is subset-minimal Herbrand models, we again refer to [16] and references therein for details.

ASP Solvers typically first compute (a subset of $ground(\Pi)$, and then use a DPLL-like branch and bound algorithm is used to find answer sets for this ground program. There are various solvers [17,18] for ASP problem specifications, we use *clasp* [3] for our experiments herein (cf. Section 5), as one of the most efficient implementations available.

As syntactic extension, in place of atoms, *clasp* allows set-like *choice expressions* of the form $E = \{A_1, \ldots, A_k\}$ which are true for any subset of E; that is, when used in heads of rules, E generates many answer sets, and such rules are often referred to as *choice rules*. For instance, $\Pi_4 = \{lights\_on.\{shop\_open, door\_locked\}\text{:-}lights\_on.\}$ has both answer sets of Π_3 plus the answer set $\{lights\_on\}$. Note that in the presence of choice rules, answer sets are not necessarily subset-minimal, we refer to [3] for details.

Another extension supported in *clasp* are optimisation statements [3] to indicate preferences between possible answer sets:

$$\#minimize\ \{A_1 : Body_1 = w_1, \ldots, A_m : Body_m = w_m\}$$

associates integer weights (defaulting to 1) with atoms A_i (conditional to $Body_i$ being true), where such a statement expresses that we want to find only answer sets with the smallest aggregated weight sum; again, variables in $A_i : Body_i = w_i$ are replaced at grounding w.r.t. all possible instantiations.

Finally, many problems conventiently modelled in ASP require a boundary parameter k that reflects the size of the solution. However, often in problems like planning or model checking this boundary (e.g., the plan length) is not known upfront, and therefore such problems are addressed by considering one problem instance after another while gradually increasing this parameter k. However, re-processing repeatedly the entire problem is a redundant approach, which is why incremental ASP (iASP) [3] natively supports incremental computation of answer sets; the intuition is rooted in treating programs in program slices (extensions). In each incremental step, a successive extension of the program is considered where previous computations are re-used as far as possible.

An iASP program is a triple $(B, A[k], Q[k])$, where B describes *static* knowledge, and $A[k]$ and $Q[k]$ are ASP programs parameterized by the incremental parameter $k \in \mathbb{N}^+$. In the iterative answer set computation of iASP, while the knowledge derived from the rules in $A[k]$ accumulates as k increases, the knowledge obtained from $Q[k]$ is only considered for the latest value of k. $A[k]$ and $Q[k]$ are called *cumulative* knowledge and *volatile* knowledge, resp. More formally, an iASP solver computes in each iteration i

$$\Pi[i] = B \cup \bigcup_{1 \leq j \leq i} A[k/j] \cup Q[k/i]$$

until an answer set for some (minimum) integer $i \geq 1$ is found. We will demonstrate next, how iASP can be successfully used to model and solve various variants of resource allocation problems in business process management.

4 Resource Allocation with iASP

For tackling the problem of resource allocation in business processes, we have developed a modular iASP program consisting of three layers. The bottom layer is the generic iASP encoding Π_N for finding a firing sequence between initial and goal markings of a 1-safe Petri net N. This provides a marking of N at each value of parameter k. On a second layer we extend Π_N towards Π_T to encode timed Petri Nets, i.e., we support business processes encoded as timed Petri nets whose activities can have a duration. Consequently, this encoding cannot only compute possible markings, but also the overall duration for a firing sequence. In other words, now we also know about the value of the overall time spent time at a firing sequence of length k. In the upper layer Π_R, we include rules and constraints about resources in order to encode an iASP program that allocates activities to available resources for a certain period of time.

Please, note some general assumptions that we make about the structure of a resource allocation problem: (i) no resource may process more than one activity at a time; (ii) each resource is continuously available for processing; (iii) no pre-emption, i.e., each activity, once started, must be completed without interruptions; and (iv) the processing times are independent of the schedule, and they are known in advance. These assumptions are common in related approaches [1].

4.1 Π_N : A Generic Formulation of 1-safe Petri Nets

Based on the notions introduced in Section 3, we formalise the firing dynamics of 1-safe Petri net $N = \langle P, T, F, M_0 \rangle$ in an iASP program $(B_N, A_N[k], Q_N[k])$. Given a goal state M_k, which for the sake of simplicity we assume to be defined in terms of a single goal place p_g, the aim is to find a shortest possible firing sequence $\overrightarrow{\sigma} = \langle t_1 t_2 ... t_k \rangle$ that does not violate the constraints, from M_0 to M_k.

B_N: $N = \langle P, T, F, M_0 \rangle$ is represented using predicates $\texttt{inPlace}_N(\texttt{p}, \texttt{t})$ and $\texttt{outPlace}_N(\texttt{p}, \texttt{t})$ that encode F. We encode different instances i of N by the predicate $\texttt{instance}_N$, which allows us to run the allocation problem against different instances of the same process; initial markings of instance M_{0i} are defined via predicate $\texttt{tokenAt}_N(\texttt{P}_0, \texttt{k}_0, \texttt{i})$ where for each $p \in P_0$, $M_0(p) = 1$.[4]

$A_N[k]$: is shown in Fig. 2. Rule (2) guesses all subsets of possible firing actions for each instance of N. Constraint (3) ensures that any transition $t \in \texttt{T}$ is fired only if all input places in $\bullet t$ have tokens. Rule (4) models the effect of the action `fire` on output places by assigning a token to each output place in the step following the firing. Constraint (5) prohibits concurrent firings of transitions $t \in p\bullet$. Rules (6) and (7) preserve tokens at place p in successive steps if none of the transitions $t \in p\bullet$ fires.

$Q_N[k]$: Finally, constraint (8) in Fig. 2 enforces a token to reach the goal place p_g (for all instances $i \in I$). The computation ends as soon as this constraint is not violated in an iteration of the iASP program, i.e., it computes the minimally necessary number of iterations k to reach the goal state.

[4] Since in the following we only consider instances of the same Petri Net, we will drop the subscript N in the predicates.

$A_N[k]$:

$\{\texttt{fire(T,k,I)} : \texttt{inPlace(P,T)}, \texttt{instance(I)}\}.$ (2)

$\texttt{:-fire(T,k,I)}, \texttt{instance(I)}, \texttt{inPlace(P,T)}, \texttt{not tokenAt(P,k,I)}.$ (3)

$\texttt{tokenAt(P,k,I):-fire(T,k-1,I)}, \texttt{outPlace(P,T)}, \texttt{instance(I)}.$ (4)

$\texttt{:-inPlace(P,T1)}, \texttt{inPlace(P,T2)}, \texttt{T1!=T2}, \texttt{fire(T1,k,I)}, \texttt{fire(T2,k,I)}, \texttt{instance(I)}.$ (5)

$\texttt{consumeToken(P,k,I):-inPlace(P,T)}, \texttt{fire(T,k,I)}, \texttt{instance(I)}.$ (6)

$\texttt{tokenAt(P,k,I):-tokenAt(P,k-1,I)}, \texttt{not consumeToken(P,k-1,I)}.$ (7)

$Q_N[k]$:

$\texttt{:-not tokenAt}(\texttt{p}_\texttt{g}, \texttt{k}, \texttt{I}), \texttt{instance(I)}.$ (8)

Fig. 2: 1-safe Petri net formulation in iASP

4.2 Π_T : Activity Scheduling using Timed Petri Net

In order to model activity durations, we extend the above iASP encoding towards Timed Petri nets: that is, Π_N is enhanced with the notion of time in Π_T. By doing so, $\Pi_N \cup \Pi_T$ becomes capable of scheduling activities in instances of a timed Petri net N_T.

B_T: We expand the input of Π_N with facts related to time and with the rules that are independent from the parameter k. For each fact $\texttt{tokenAt}(\texttt{p}_0, \texttt{k}_0, \texttt{i})$ previously defined we add in B_T a fact $\texttt{timeAt}(\texttt{p}_0, \texttt{c}_0, \texttt{k}_0, \texttt{i})$ where c_0 is the initial time at p_0. In order to distinguish activity transitions and ("silent") non-activity transitions[5], we add facts `activity(t)` for all activities. Durations of activities are specified with facts `timeActivity(t,c)` where t is an activity and $c \in \mathbb{Z}^+$. The remainder of B_T is given by rules (9,10) in Fig. 3: rule (9) defines firing delays of each transition in N and rule (10) assigns duration zero to activity transitions per default, where the delay is not otherwise specified.

$A_T[k]$: Rule (13) defines the effect of action `fire` on `timeAt` for all output places $t\bullet$ where t is a *non-activity* transition. In this case, the maximum time among the input places, which is computed by rules (11,12), is propagated over all output places. As opposed to (13), rule (14) defines the effect of action `fire` on `timeAt` for *activity* transitions. Time value derived in rule (14) for the next step is the sum of the maximum time value at the input places and the value of the activity duration. Rule (15) conserves the time value of a place in the succeeding step k in case the transition does not fire at step $k-1$.

$Q_T[k]$: On top of $Q_N[k]$, an optimization statement (16) is added for computing answer sets with the minimum time cost.

4.3 Π_R : Resource Allocation

In the last layer of our iASP program, Π_R, we additionally formalise resources and related concepts. $\Pi_N \cup \Pi_T \cup \Pi_R$ allow allocating resources to activities for a time optimal execution of all defined instances of N_T.

[5] Recall: in Petri nets representing business processes, activity transitions are empty squares while silent transitions are represented in filled squares (cf. Fig. 1a).

B_T :

```
firingDelay(T,C):-timeActivity(T,C).                                          (9)
firingDelay(T,0):-not timeActivity(T,_),activity(T).                          (10)
```

$A_T[k]$:

```
greTimeInPlace(P1,T,k,I):-inPlace(P1,T),inPlace(P2,T),fire(T,k,I),           (11)
                          timeAt(P1,C1,k,I),timeAt(P2,C2,k,I),P1!=P2,
                          C1<C2,instance(I).
maxTimeInPlace(P,T,k,I):-inPlace(P,T),not greTimePlace(P,T,k,I),             (12)
                         fire(T,k,I),instance(I).
timeAt(P2,C,k,I):-not activity(T),fire(T,k-1,I),outPlace(P2,T),              (13)
                  maxTimeInPlace(P,T,k-1,I),timeAt(P,C,k-1,I),
                  instance(I).
timeAt(P2,C1,k,I):-activity(T),fire(T,k-1,I),outPlace(P2,T),                 (14)
                   maxTimeInPlace(P,T,k-1,I),timeAt(P,C,k-1,I),
                   firingDelay(T,D),C1=C+D,instance(I).
timeAt(P,C,k,I) :-not consumeToken(P,k-1,I),inPlace(P,T),                    (15)
                  timeAt(P,C,k-1,I),instance(I).
```

$Q_T[k]'$:

```
#minimize{timeAt(p_g,C,k,I):instance(I)=C}                                   (16)
```

Fig. 3: Scheduling extension

***B_R*:** The facts related to resources and organisational models are defined in the input of Π_T. An example organisational model is shown in Fig. 1b. Facts `hasRole(r,l)` relates a resource r to a role l. Activities are related to a role via facts of the form `canExecute(l,t)`, which means that a role l is allowed to performing an activity t. An optional estimated duration for a resource to execute an activity can be defined by `timeActivityResource(t,r,c)`. Similarly an optional estimated duration for a role per activity can be defined by `timeActivityRole(t,l,c)`. Both can override the default `timeActivity(t,c)`. In particular, the order ($>$) preferred in resource-time allocation is `timeActivityResource` $>$ `timeActivityRole` $>$ `timeActivity`. This is especially useful when a resource or a role is known to execute a particular activity in a particular amount of time, which can be different from the default duration of the activity. In our program (cf. Fig. 4) this preference computation is encoded in rules (17-21). Rules (17,18) are projections of optionally defined activity execution durations. Rules (19-21) derive correct execution duration for resource-activity pairs considering both mandatory and optional durations.

***$A_R[k]$*:** In the iterative part, rule (22) allocates a resource r to an activity t from time c to time $c2$. Note that, for handling optional execution durations, rule (14) from Fig. 3 is replaced by rule (14)*. Rule (23) along with constraint (24) prohibits any firing of an activity transition that is not allocated to a resource. Constraint (25) ensures that an activity cannot be assigned to more than one

B_R :

```
existsTimeActivityResource(T,R):-timeActivityResource(T,R,C).                    (17)
existsTimeActivityRole(T,L):-timeActivityRole(T,L,C),hasRole(R,L).               (18)
takesTime(T,R,C):-timeActivityResource(T,R,C).                                   (19)
takesTime(T,R,C):-timeActivityRole(T,L,C),hasRole(R,L),canExecute(L,T),          (20)
                  not existsTimeResource(T,R).
takesTime(T,R,C):-firingDelay(T,C),hasRole(R,L),canExecute(L,T),                 (21)
                  not existsTimeActivityResource(T,R),
                  not existsTimeActivityRole(T,L).
```

$A_R[k]$:

```
{assign(R,T,C,C2,k,I) : takesTime(T,R,C),C2 = C + D}:-inPlace(P1,T),             (22)
                        timeAt(P1,C,k,I),activity(T),instance(I).
timeAt(P2,C2,k,I):-activity(T),assign(R,T,C1,C2,k − 1,I),                        (14)*
                   fire(T,k − 1,I),outPlace(P2,T),instance(I).
assigned(T,k,I):-assign(R,T,C1,C2,k,I).                                          (23)
:-not assigned(T,k,I),fire(T,k,I),activity(T),instance(I).                       (24)
:-assign(R,T,C1,C2,K,I),assign(R1,T,C3,C4,K,I),R! = R1.                          (25)
:-assign(R,T1,C1,C2,K1,I1),assign(R,T2,C1,C2,K2,I2),C1! = C2,T1! = T2.           (26)
:-assign(R,T,C1,C2,K1,I1),assign(R,T,C1,C2,K2,I2),C1! = C2,I1! = I2.             (27)
:-assign(R,T1,C1,C2,K1,I1),assign(R,T2,C1,C2,K2,I2),                             (28)
  C1! = C2,I1! = I2,T1! = T2.
:-assign(R,T,B1,B2,K1,I),assign(R,T2,A1,A2,K2,I2),A1 > B1,A1 < B2.               (29)
:-assign(R,T,B1,B2,K1,I),assign(R,T2,A1,A2,K2,I2),A2 < B2,A2 > B1.               (30)
```

Fig. 4: Allocation extension

resource. Constraints (26-28) guarantee that only one resource is assigned to one activity at a time. Constraints (29,30) prevents a busy resource to be re-assigned.

Time Relaxation In case a resource is busy at the time when s/he is required for another activity, our program would be unsatisfiable as it is. We add rules (31) and (32) (cf. Fig. 5) into $A_T[k]$ for allowing the demanding activity to wait until the required resource is available again.

5 Evaluation

We demonstrate the applicability and effectiveness of the proposed computational method for resource allocation in business processes by using it with a specific process. In order to measure performance and scalability, we conduct a batch experiment using generated examples of timed Petri nets of different sizes.

$A_T[k]'$:

```
relaxationAt(P,C + 1,k,I):-timeAt(P,C,k − 1,I),inPlace(P,T),activity(T),  (31)
                           not consumeToken(P,k − 1,I),instance(I).
timeAt(P,C,k,I):-relaxationAt(P,C,k,I).                                   (32)
```

Fig. 5: Time relaxation for optimality

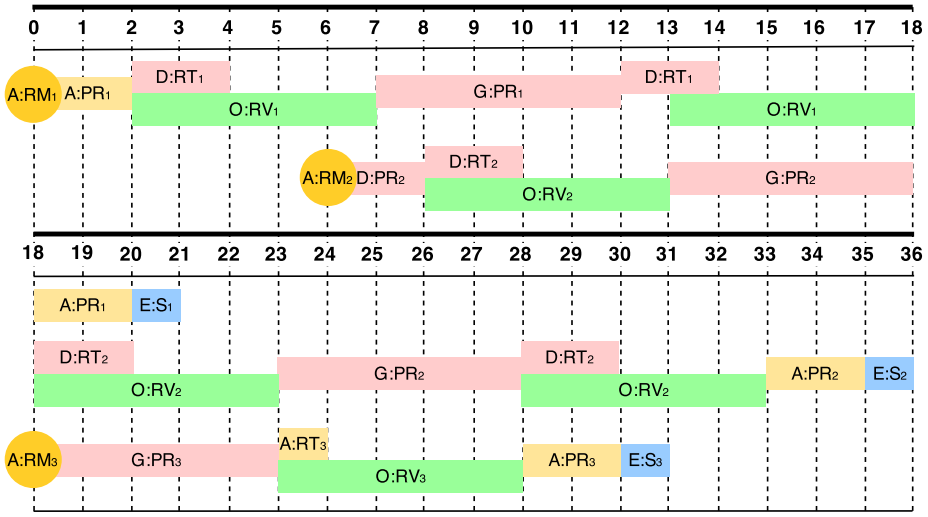


Fig. 6: Instance I - 2 loop repetitions / instance II - 3 loop repetitions / instance III - 1 loop repetition

5.1 Example Scenario

We apply our method to a business process model that specifies the process of publishing a book as described in Section 2.1. The input of the program encoded in ASP following the explanations in Section 4 is: (i) three different instances $i1$, $i2$, $i3$ of the timed Petri net depicted in Fig. 1a, whose starting times are defined as $t_{0i1} = 0$, $t_{0i2} = 6$ and $t_{0i3} = 11$, respectively; (ii) the organisational model and optional activity times for resources and roles as shown in Fig. 1b, (iii) role-activity relation defined in Section 2. We also add additional constraints for enforcing the firing sequence to go through the loop present in the process two, three and one times for $i1$, $i2$ and $i3$, respectively.

The computed optimal resource allocation is visualised in Fig. 6. The allocation periods are depicted as coloured rectangles with a tag on it. Each tag has three parts: an initial with the initials of a resource, a short version of the allocated activity name and a subscript representing the instance ID. For example, $D : PR_1$ means that *Drew* is allocated to activity *Proofreading*. The colours of these rectangles correspond to the colours used for the roles depicted in Fig. 1b. Note that *Amy* has more than one roles in the organisation.

The longest process instance $i2$ finishes in 36 time units. Several solutions were found for that global minimum time. In Fig. 6, instances 1 and 2 finish without interruptions. However, instance 3 waits 7 time units for the availability of *Glen* to start performing activity *Proofread*, since he is busy performing that activity for process instance 2 until time unit 23. All-in-all, this computation optimises the use of resource *Oliver*, who is the only *Graphic Artist* and is required in all the process instances. Please, note that, e.g., in instance 2 *Drew* is selected to perform activity *Proofread* because it takes him only 2 time units (cf. Fig. 1b), half of the default duration associated with the activity (cf. Fig. 1a). This responds to the preference order described in Section 4.3.

id	$\|I\|$	$\|L\|$	$\|f(T)\|$	k	s	m
1	1	1	10	8	1.13	10.2
2	2	1	20	21	7.38	72.2
3	3	1	38	9	176.45	432.1
4	1	2	10	3	0.57	0
5	2	2	20	21	83.03	459.4
6	3	2	42	31	199.46	756.8
7	1	3	10	11	1.27	17.9
8	2	3	20	16	28.57	229
9	3	3	38	21	85.73	475.1

id	$\|I\|$	$\|L\|$	$\|f(T)\|$	k	s	m
10	1	1	24	4	15.02	101.6
11	2	1	48	25	90.87	419
12	3	1	72	33	193.72	372.9
13	1	2	28	29	33.96	186.2
14	2	2	60	7	1314.73	2877.2
15	3	2	n/a	n/a	10800	5744.1
16	1	3	24	25	17.5	83.9
17	2	3	48	28	161.15	496.5
18	3	3	96	4	2366.24	4473.9

Table 1: Experiments: (1-9) Loops not enforced, (10-18) Loops enforced

5.2 Performance

For our experimental evaluation, we generated a set of sound choice-free timed Petri nets (cf. Section 3). We varied the number of existing loops in these Petri nets and the number of parallel process instances. We use the same organisational model for all of the generated Petri nets, specifically the one depicted in Fig. 1b. We performed these experiments on a Linux server (4 CPU cores/2.4GHz/32GB RAM). *clasp* was used as ASP solver with the multi-threading mode enabled.

The results are shown in Table 1 in two parts. In the programs on the left hand side (1-9), no transitions in the loops are enforced to be fired. In the programs on the right hand side (10-18), each loop in the Petri net is constrained to be followed at least one time. The columns of the table are as follows: id is the identifier of a generated program, $|I|$ is the number of parallel instances, $|L|$ is the number of loops, $|f(T)|$ is the number of fired transitions from initial to goal state, k is the final value of that parameter, s is the time in seconds to find the answer set of the program, and m is the maximum memory usage in megabytes.

For instance, it takes the solver 1.13 seconds to find an answer set for a Petri net with one loop that is not enforced at run time, and 15.02 seconds for a similar Petri net in which the loop is executed. This is satisfactory for many planning scenarios with large processes, as they can be scheduled in a few seconds/minutes and executed for a long period of time.

6 Conclusions and Future Work

We have introduced an approach for automated resource allocation in business processes that relies on ASP to find an optimal solution. The result is a work distribution (i.e., an activity allocation) that ensures that all the process activities can finish in the minimum amount of time given a set of resources. Unlike similar approaches, it is capable of dealing with cyclic processes and concurrent process instances as our encoding in ASP is flexible and extensible. Note that extensions like constraints enforcing separation and binding of duties [19] can be easily added in our formalism, which we omitted due to space restrictions.

We plan to conduct further performance measurements and compare them to other formalisms, e.g., constraint solvers. We are confident that there is room

for optimisations (e.g., symmetry breaking [4] or similar techniques) that have been successfully applied in ASP.

References

1. W.M.P. van der Aalst. Petri net based scheduling. *Operations-Research-Spektrum*, 18(4):219–229, 1996.
2. A. Rozinat and R. S. Mans. Mining CPN Models: Discovering Process Models with Data from Event Logs. In *Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN*, pages 57–76, 2006.
3. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
4. Andreas A. Falkner, Gottfried Schenner, Gerhard Friedrich, and Anna Ryabokon. Testing object-oriented configurators with ASP. In *Workshop on Configuration at ECAI 2012*, pages 21–26, 2012.
5. Tadao Murata. Petri nets: Properties, analysis and applications. *IEEE*, 77(4):541–580, 1989.
6. Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri Net Transformations for Business Processes - A Survey. *Transactions on Petri Nets and Other Models of Concurrency II*, 2:46–63, 2009.
7. J. Carlier and E. Pinson. An Algorithm for Solving the Job-shop Problem. *Manage. Sci.*, 35(2):164–176, February 1989.
8. Pinar Senkul and Ismail H. Toroslu. An Architecture for Workflow Scheduling Under Resource Allocation Constraints. *Inf. Syst.*, 30(5):399–422, July 2005.
9. Stefan Heinz and Christopher Beck. Solving Resource Allocation/Scheduling Problems with Constraint Integer Programming. In *COPLAS 2011*, pages 23–30, 2011.
10. Jan Weglarz. Project Scheduling with Continuously-Divisible, Doubly Constrained Resources. *Management Science*, 27(9):1040–1053, 1981.
11. M.H.A. Hendriks, B. Voeten, and L. Kroep. Human resource allocation in a multi-project R&D environment: Resource capacity allocation and project portfolio planning in practice. *Int. J. of Project Management*, 17(3):181–188, 1999.
12. Yann Chevaleyre, Paul E. Dunne, Ulle Endriss, Jrme Lang, Michel Lematre, Nicolas Maudet, Julian Padget, Steve Phelps, Juan A. Rodrguez-aguilar, and Paulo Sousa. Issues in multiagent resource allocation. *Informatica*, 30:2006, 2006.
13. Chongjie Zhang, Victor Lesser, and Prashant Shenoy. A Multi-Agent Learning Approach to Online Distributed Resource Allocation. In *International Joint Conference on Artificial Intelligence (IJCAI'09)*, volume 1, pages 361–366, 2009.
14. Yuhong Yan, Z. Maamar, and Weiming Shen. Integration of workflow and agent technology for business process management. In *Computer Supported Cooperative Work in Design*, pages 420–426, 2001.
15. Louchka Popova-Zeugmann. Time Petri Nets. In *Time and Petri Nets*, pages 139–140. Springer Berlin Heidelberg, 2013.
16. Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
17. Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. The Design of the Fifth Answer Set Programming Competition. *CoRR*, 2014.
18. Marijn JH Heule and Torsten Schaub. What's Hot in the SAT and ASP Competitions. In *AAAI*, 2015.
19. Maria Leitner and Stefanie Rinderle-Ma. A systematic review on security in Process-Aware Information Systems Constitution, challenges, and future directions. *Information and Software Technology*, 56(3):273 – 293, 2014.