Science of Computer Programming 225 (2023) 102896

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Cost analysis for a resource sensitive workflow modelling language [☆]

Muhammad Rizwan Ali [*], Yngve Lamo, Violet Ka I Pun [*]

Western Norway University of Applied Sciences, Norway

ARTICLE INFO

Article history:
Received 20 May 2022
Received in revised form 17 October 2022
Accepted 1 November 2022
Available online 9 November 2022

Keywords:
Cross-organisational workflows
Resource planning
Formal modelling
Static analysis

ABSTRACT

Workflow analysis usually requires domain-specific knowledge from the domain experts, making it a relatively manual process. In addition, workflows often cross organisational boundaries. As a result, minor local modifications in the workflow of a collaborative partner may be propagated to other concurrently running tasks of the workflow, which is difficult for the domain experts to recognise since they only have a limited (local) view of the workflow. Therefore, changes in cross-organisational workflows may result in significant adverse impacts. This paper presents a resource-sensitive formal modelling language, $\mathcal{R}$PL, which has explicit notions of task dependencies, qualitative assessment of resources, time advancement and method execution deadlines. The language allows the workflow analysers to estimate the effect of changes in collaborative workflows with respect to cost in terms of execution time. This paper proposes a static analysis to compute the worst execution time of a cross-organisational workflow modelled in $\mathcal{R}$PL by defining a compositional function that translates an $\mathcal{R}$PL program to a set of cost equations.

1. Introduction

Workflow management can be seen as an effective method of monitoring, managing, improving and analysing business processes using IT assistance [1]. Workflow management systems (WMS) automate business processes and play a key role in collaborative business domains such as supply chain management and customer relationship management. As a result, WMS is regarded as among the most effective systems for facilitating cooperative business operations [16].

With the fast growth of e-commerce and virtual companies, corporations frequently work beyond organisational borders, engaging with others to meet competitive challenges. Moreover, the rapid growth of the Internet and digital technology encourages collaboration across widely distant businesses [40]. The adoption of cross-organisational workflows allows restructuring of business processes beyond the limits of an organisation [2]. Cross-organisational workflows often comprise multiple concurrent workflows running in various departments within the same organisation or different organisations, and sometimes share resources. Examples are the workflows of a hospital's emergency department, outpatient department, and pathology department.

Organisations very often analyse their workflows using experts with domain specific knowledge (the analysts) to ensure optimal resource allocation, task management and workflow updates to cope with the competitors. Nonetheless, analysing

[☆] This work is part of the *CroFlow project: Enabling Highly Automated Cross-Organisational Workflow Planning, funded by the Research Council of Norway (grant no. 326249).*

[*] Corresponding authors.
E-mail addresses: mral@hvl.no (M.R. Ali), Yngve.Lamo@hvl.no (Y. Lamo), Violet.Ka.I.Pun@hvl.no (V.K.I Pun).

https://doi.org/10.1016/j.scico.2022.102896
0167-6423/

cross-organisational workflows from a global perspective can be incredibly challenging as the analysts may only have limited domain knowledge of their local workflows, may not have a common understanding of all the collaborative workflows, shared resources, and across-workflow task dependencies. Additionally, modifying cross-organisational workflows is error-prone: one modification in a workflow may result in significant changes in other concurrently running workflows, and a minor mistake might have significant (negative) consequences, particularly in domains like healthcare and aviation.

Workflows have been significantly digitalised and automated using the most popular modelling languages. Furthermore, many formal approaches have been extensively utilised to formalise and analyse workflows. A detailed comparison of some most popular languages and techniques is presented in Section 2. However, as per our knowledge, cross-organisational workflow analysis remains a somewhat manual process as current techniques and tools often lack domain-specific knowledge to support the automation of workflow analysis and updates. Therefore, there is a need for a formal approach that supports modelling of cross-organisational workflows, including cross-workflow task dependencies and shared resource allocation. Also, the required approach should allow the analysts to simulate changes in the design of workflows and understand the effect of the changes in all collaborating workflows before the actual implementation and execution of updated workflows.

This paper presents a resource-sensitive formal modelling language $\mathcal{R}$PL, which can be used to model cross-organisational workflows formally. The language supports the creation, acquisition and release of shared resources, allows specifying inter-workflow task dependencies. Additionally, the language has a unique notion of time consumption and supports specifying the deadlines of task execution. It also enables a modeller to couple collaborative workflows through shared resources and task dependencies, and to create a common knowledge base of all collaborating workflows in the form of shared resources. In addition, we present an analysis based on the work in [31] to statically over-approximate the worst execution time of the cross-organisational workflows modelled as an $\mathcal{R}$PL program by translating the program into a set of cost equations that can be fed to an off-the-shelf constraint solver (e.g., [19,5]). This enables analysts to estimate the effects of the workflow (and its possible changes) in terms of execution time before the actual implementation.

A preliminary idea of the language was presented in [8,9], where resources are assessed by quantity, and the conditional statement supports only one-way selection. As compared to [8,9], this paper presents the language which assesses resources by both quantity and quality, the conditional statement supports two-way selection, the methods can be invoked with deadlines, and task dependencies may have logical disjunction between them. The language and the cost analysis can help facilitate planning cross-organisational workflows and may ultimately contribute to automated planning.

The rest of the paper is organised as follows: Section 2 presents a comparative analysis of the most popular workflow modelling approaches, and briefly discusses the work related to static cost analysis. Section 3 introduces the syntax and semantics of the $\mathcal{R}$PL language. Section 3.3 presents a motivating example of collaborative workflows modelled in $\mathcal{R}$PL. Section 4 shows a static analysis to over-approximate the execution time of an $\mathcal{R}$PL program. Section 5 shows the correctness of the analysis. Finally, we summarise the paper and discuss possible future work in Section 6.

2. Related work

This section first presents some of the most widely studied workflow modelling approaches in scientific literature and practice scenarios, and presents a comparative analysis of the selected approaches with $\mathcal{R}$PL. We then discuss some of the work that is related to static cost analysis.

2.1. Workflow modelling approaches and practices

Workflows can be modelled using graphical or textual modelling languages or a combination of both. Selecting an appropriate formalism for modelling and analysing cross-organisational workflows is still an open issue. Several approaches for workflow modelling already exist, some have a strict formal foundation, and others are tool based with an unclear foundation. However, it is not yet clear which is the most valuable, nor if they are more or less useful in particular contexts. Simple workflows can be modelled solely through a graphical designer with some software products. Such systems rely on capturing the information relevant to the workflow process through a user-friendly interface aimed at non-programmers and then compiling that information into practical workflows. However, complex, time-sensitive, or cross-organisational workflows are hard to represent with informal modelling languages [23]. Therefore, the demand for employing a formal modelling language emerges. The reason is that formal modelling languages are more rigorous and explore every possibility to ensure completeness and correctness.

There are several aspects of formal modelling languages that need to be enhanced in order to support cross-organisational workflows. Some of the primary aspects are as follows:

1. Workflow Description: It should provide the constructs to model business rules, including control flow and reuse of common processes without recreating them repeatedly.
2. Communication Mechanism: It should support some communication mechanism to share data and information between the tasks of one workflow as well as the tasks of different collaborating workflows.
3. Resource Modelling: It should explicitly specify resources that are often shared between collaborative partners.
4. Knowledge Representation: It should allow the representation of local as well as global knowledge.

Table 1
Definition of concepts.

Concepts			Definition
Workflow Description	(1)	Task	A task is a sequence of steps that need to be executed.
	(2)	Control Flow	Control flow is an order in which a task executes.
	(3)	Sub-Workflow	It is splitting up a complex workflow into smaller workflows that are more manageable and easier to understand. By creating sub-workflows, common processes can be reused without being recreated.
Communication Mechanism	(3)	Intra-Workflow Communication	Intra-workflow communication happens between the participants of a single workflow.
	(4)	Inter-Workflow Communication	Inter-workflow communication happens between the participants of different workflows.
Resource Modelling	(5)	Participant	The participant is a type of resource who can carry out actions.
	(6)	Material Resources	A material resource (tangible resource) is anything that has actual physical existence and is owned by an individual participant or company, like equipment or vehicles.
	(7)	Internal Participants	Internal participants are those who collaborate within a single workflow.
	(8)	External Participants	External participants are those who collaborate across different workflows.
	(9)	Shared Resources	The resources which all the workflows can access and share.
Knowledge Representation	(10)	Tacit Knowledge	Tacit knowledge is a piece of implicit knowledge tied to the human (in his brain), such as experience, skills, talents and abilities.
	(11)	Explicit Knowledge	Explicit knowledge is a formal knowledge that can be expressed in words, mathematical expressions, specifications, or computer programs and easily shared with others.
	(12)	External Knowledge	External knowledge is a piece of knowledge created and stored within the boundaries of other organisations.
Non-Functional Aspects	(13)	Executable	An executable workflow is one whose execution can be simulated by a computer program.
	(14)	Approach	It can be knowledge-oriented, formal and informal.
	(15)	Usage	Usage refers to where the given approach is used, such as in industry and academia.
	(16)	Complexity	The ability to describe the advanced control flows and intra as well as inter-workflow collaboration.
	(17)	Expressiveness	The power and suitability to integrate and specify all business process aspects.
	(18)	Understand-ability	The degree to which the workflow formalism and model can be easily interpreted by all stakeholders in the organization.
	(19)	Tools Support	The availability of diversified set of tools supporting the formalism and transformation to other formats to benefit from models interchange among tools.

5. Non-functional Aspects: It should be executable, more expressive, easy to understand, have some tool support, have formal semantics and be analysable.

Table 1 defines these aspects in detail. There are many approaches for workflow modelling; however, we have selected some of the most prominent ones, considering the aspects defined in Table 1. The selected approaches are classified according to the three most prominent categories for workflow representation, some of them follow a process-oriented approach (UML-AD [17], RAD [34], BPMN [13], BPEL [35], eEPC [38] and DWM [32]), some follow a knowledge-oriented approach (PROMOTE [39], Oliveira [33] and KMDL [21]) and others adopt a formal approach (CPN [27], YAWL [3], Pi-Calculus [36] and Timed-Automata [41]).

Table 2 shows a comparison of $\mathcal{R}$PL with the selected approaches. The concepts of workflow description, resources, knowledge and communication are compared with an evaluation scale, ×, –, and ✓, where × is an evaluation that the concept is not supported at all (notation element is missing, the fact is not illustratable), – partially supported (notation element is missing, the fact is illustratable), and ✓ fully supported (notation element is available, the fact is illustratable). For non-functional aspects, we use different evaluation scales. To indicate executability, we use ✓ and ×. To indicate different approach, we use P, K and F, where P refers to process-oriented, K knowledge-oriented and F formal approach. We use I and A to indicate usage in industry and academia, respectively. Moreover, a 4-point scale from 0 to 3 is used for complexity, expressiveness, understandability and tools support. This scale assigns a score 0 to any requirement that the language cannot support. It assigns score 1 if partially supported, 2 if satisfactorily supported and 3 if it is very well supported.

In the following, we discuss the languages under comparison from the five different aspects mentioned above.

Workflow Description. As shown in Table 2, all the selected languages are task-oriented. However, most languages lack control flow. While UML-AD compares well to existing WMS, the language does not fully capture advanced synchronisation patterns, e.g., N-out-of-M joins [12]. RAD is expressive enough to model workflows; however, it lacks sub-workflow support [24]. PROMOTE, Oliveira, and KMDL are developed for knowledge representation. They support the three essential control flow elements AND, OR, and XOR operators, but have shortcomings concerning their ability of representing complex decisions and data flows. Moreover, these languages do not support decomposing a complex workflow into sub-

Table 2
Comparison of approaches, where (i) correspond to the numbered items in Table 1.

Languages	Workflow Description			Comm. Mechanism		Resource Modelling				Knowledge Representation			Non-Functional Aspects						
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)
BPMN	✓	✓	✓	✓	–	✓	✓	-	✓	×	–	–	×	P	I	2	3	3	3
UML-AD	✓	–	✓	×	×	–	–	×	–	×	×	×	×	P	I	1	1	2	3
RAD	✓	–	–	–	×	–	✓	×	–	×	×	×	×	P	A	1	1	2	1
BPEL	✓	✓	✓	✓	×	✓	✓	–	✓	×	–	×	✓	P	I	2	2	1	2
DWM	✓	✓	✓	✓	×	✓	✓	–	–	×	–	×	✓	P	A	3	2	1	1
eEPC	✓	✓	–	×	×	–	–	×	–	–	–	×	✓	P	I	1	2	2	1
PROMOTE	✓	–	×	×	×	–	✓	×	×	–	–	×	×	K	A	1	1	2	0
Oliveira	✓	–	×	✓	×	–	✓	×	×	✓	–	×	×	K	A	1	1	2	0
KMDL	✓	–	×	–	×	–	✓	×	–	✓	–	×	✓	K	I	2	2	2	1
CPN	✓	–	✓	✓	–	✓	✓	–	–	×	-	×	✓	F	I	2	2	2	2
YAWL	✓	✓	✓	✓	–	✓	–	–	–	×	–	×	✓	F	I	2	3	2	2
Pi-Calculus	✓	✓	✓	✓	–	✓	✓	–	–	×	–	×	✓	F	I	2	2	1	2
Timed-Automata	✓	✓	✓	✓	–	✓	✓	–	–	×	–	×	✓	F	I	2	2	2	2
$\mathcal{R}$PL	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	F	I	3	2	1	0

workflows. While high-level Petri nets (CPN) outperform most existing languages, the control flow modelling is not entirely satisfactory, especially for patterns including multiple instances or advanced synchronisation patterns [3]. Like BPMN, BPEL, YAWL, Pi-Calculus and Timed-Automata, $\mathcal{R}$PL is expressive enough from a workflow description perspective because $\mathcal{R}$PL offers explicit notations for the organisation of control flows, complex decisions, exclusive event-based decisions and parallel event-based decisions.

Communication Mechanism. From the communication perspective, most languages support sending or receiving messages within one workflow, but not UML-AD, eEPC and PROMOTE. For inter-workflow communication, BPMN allows exchanging messages among different workflows; however, it does not provide the semantics to depict the dependencies of the global control flow of the message exchange [12]. Moreover, inter-workflow communication is illustrated to some extent with the help of hierarchical models in CPN, sub-processes in Pi-Calculus and templates in Timed-Automata. Compared to these approaches, $\mathcal{R}$PL uses the actor model of concurrency and cooperative scheduling (safe concurrency). Additionally, $\mathcal{R}$PL supports inter-workflow communication by joining different workflow models using explicit notions of shared resources and task dependencies, considering the dependencies of the global control flow of the message exchange.

Resource Modelling. Almost all selected languages support the modelling of material resources and internal participants; however, UML-AD, RAD, PROMOTE, Oliveira and KMDL do not support external participants. Apart from PROMOTE and Oliveira, all languages support shared resources to some extent. Compared to them, $\mathcal{R}$PL fully supports external participants and shared resources as $\mathcal{R}$PL have explicit notions for inter-workflow task dependencies and resources that can be shared safely (without deadlock) between cross-organisational workflows.

Knowledge Representation. From the knowledge perspective, KMDL, Oliveira and PROMOTE are adequate for modelling tacit and explicit knowledge categories. On the other hand, eEPC, which belongs to the traditional process-oriented formalism, has to be adapted for knowledge-based modelling. In eEPC, knowledge is represented by two object types, knowledge category and documented knowledge, represented by knowledge structures and knowledge maps. However, eEPC models lack personal references and knowledge transformations [38]. On the contrary, availability and knowledge transformations can more easily be demonstrated indirectly with the PROMOTE notation [39]. The tacit knowledge of a person could be used for creating a skilled catalogue. Oliveira and KMDL support modelling tacit knowledge, which belongs to a particular person or group. Explicit knowledge can be modelled with documented information such as messages, documents and records. Besides UML-AD and RAD, all selected languages support the modelling of explicit knowledge but lack external knowledge. In contrast, $\mathcal{R}$PL supports the creation of a shared knowledge base in terms of shared resources and supports sharing knowledge across organisations.

Non-functional Aspects. Among the compared languages, BPMN, UML-AD, RAD, BPEL, DWM and eEPC are process-oriented; PROMOTE, Oliveira and KMDL are knowledge-oriented; and only CPN, YAWL, Pi-Calculus, Timed-Automata and $\mathcal{R}$PL have formal semantics. Most of the selected languages are executable and have tools support for simulation. DWM uses DynaFlow [32] for execution. eEPC can be modelled and executed in ARIS Toolset [30]. CPN Tools [28] support editing, simulating and analysing of CPN workflows. YAWL is a free, open-source workflow system [26]. Workflows modelled in Pi-Calculus can be simulated in PiVizTool [11]. UPPAAL [10] is a tool for modelling, simulation and analysis of Timed-Automata models. Though BPMN has the support of several tools, however, it needs to be translated into an XML based executable modelling language for simulation, i.e., BPEL.

In terms of workflow description, expressiveness, understandability, and tools support, BPMN is likely the best option since anyone can easily use it regardless of their background. On the contrary, the formal approaches require some program-

$$
\begin{array}{rcl@{\qquad}rcl}
P & ::= & R\ \overline{Cl}\ \{\overline{T\ x;}\ s\} & s & ::= & x = rhs \mid \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{skip} \mid \textbf{return}\ e \\
R & ::= & [r \mapsto (b, \mathcal{Q})] & & & \mid \textbf{wait}(f) \mid \textbf{cost}(e) \mid \textbf{add}(rs) \mid \textbf{release}(rid) \mid s\ ;\ s \\
Cl & ::= & \textbf{class C}\ \{\overline{T\ x;}\ \overline{M}\} & rhs & ::= & e \mid \textbf{new C} \mid \textbf{hold}(rs) \mid f.\textbf{get} \\
M & ::= & Sg\ \{\overline{T\ x;}\ s\} & & & \mid m(x, \overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e' \\
Sg & ::= & B\ m(\overline{T\ y}) & & & \mid !m(x, \overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e' \\
T & ::= & B \mid \textbf{C} \mid \textbf{Rid} \mid \textbf{Fut}\langle B\rangle & e & ::= & k \mid x \mid g \mid \textbf{this} \mid \texttt{null} \mid rid \\
B & ::= & \textbf{Int} \mid \textbf{Bool} \mid \textbf{Unit} & g & ::= & b \mid fs \mid g \wedge g \\
 & & & rs & ::= & \emptyset \mid \{\mathcal{Q}\} \cup rs \\
 & & & rid & ::= & \emptyset \mid \{r\} \cup rid \\
 & & & fs & ::= & f? \mid fs \wedge fs
\end{array}
$$

Fig. 1. Syntax of $\mathcal{R}$PL.

ming knowledge. While BPMN has fully documented syntactic rules, the existing semantics is only defined in a narrative form employing some unstable terminology [14]. Moreover, the capability to inspect the semantics correctness of models at the static time is required for modelling inter-organisational workflows. Instead, the static analysis of informal models is deprived of ambiguities in the standard specification and the language complexity. Also, it is more demanding to present formal semantics to analyse BPMN models [14].

Compared to all selected languages, $\mathcal{R}$PL has explicit notions to couple workflows through shared resources and task dependencies and supports inter-organisation workflow modelling and analysis. Moreover, in $\mathcal{R}$PL, transition rules are formally defined in the form of structural operational semantics (SOS), and $\mathcal{R}$PL is executable on the semantics level. Although currently $\mathcal{R}$PL does not support any tool for simulation, we aim in our future work to implement a framework for $\mathcal{R}$PL, where one can create, change, simulate and estimate the effect of changes in cross-organisational workflows. Therefore, $\mathcal{R}$PL can be seen a helpful tool for automating various industrial inter-organisational workflows.

2.2. Cost analysis

For static cost analysis, numerous techniques have been introduced. For example, [6] presents the first approach to the automatic cost analysis of object-oriented bytecode programs, [25] proposes the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. Our approach differs from [25] in the concurrency model and the distinction between blocking and non-blocking synchronisation.

The authors in [7] present a cost analysis that targets a language with the same concurrency model as $\mathcal{R}$PL. On the contrary, the analysis in [7] is not compositional, and it does not demand any control of synchronisation sets because it takes the entire program and computes the components that may execute in parallel. Another approach presented in [20] analyses time complexity for concurrent programs by deriving the time-consuming behaviour with a type-and-effect system. However, the analysis in [20] lacks in computing the costs of methods that have invocations to arguments (namely actors) which do not live in the same machine. Besides, [31] defines a compositional analysis for concurrent programs and overcomes the lacking of [20].

Formal method tools have also been used for worst-case execution time (WCET) analysis. UPPAAL [10] is being used to model, simulate and verify workflows modelled in timed automata. SWEET [18] can generate models using UPPAAL syntax [37]. However, the C-style UPPAAL syntax is limited when it comes to function calls; for example array arguments need to have a known size, and if larger data is later to be stored, a new array needs to be created, which makes the code less generic. Consequently, the functions in UPPAAL syntax are intended to be very small and simple [22].

Compared to the above-mentioned tools and techniques, this paper handles a more expressive language that supports modelling complex workflows and is sensitive to task dependencies and resource consumption.

3. Formal workflow modelling language $\mathcal{R}$PL

In this section, we present the formal modelling language $\mathcal{R}$PL. The language is inspired by an active object language ABS [29]. It has a Java-like syntax and actor-based concurrency model. In actor-based concurrency models [4], actors are primitives for concurrent computation. Actors can send a finite number of messages to each other, create a finite number of new actors, or alter their private states. One of the primary characteristics of actor-based concurrency models is that only one message is processed per actor, so the invariants of each actor are preserved without locks.

In addition, the language uses cooperating scheduling to control the internal interleaving of processes inside an object with explicit scheduling points. It also uses explicit notions to specify time advancement, to assign a deadline to a task (expressed as a method), and to indicate resources required for each task and dependencies between tasks.

3.1. The syntax of $\mathcal{R}$PL

The syntax of the $\mathcal{R}$PL is given in Fig. 1. An overlined element represents a (possibly empty) finite sequence of such elements separated by commas, e.g., $\overline{T}$ implies a sequence $T_1, T_2, \ldots, T_n$.

An $\mathcal{R}$PL program P comprises resources R, a sequence of class declarations $\overline{Cl}$ and a main method body $\{\overline{T\ x;}\ s\}$, where $\overline{T\ x;}$ is the declaration of local variables and s is a statement.

$$\boxed{[r_1 \mapsto (\mathsf{true},\ \{\mathsf{Doctor,Ortho,3}\}),\ r_2 \mapsto (\mathsf{true},\ \{\mathsf{Doctor,Cardio,8}\}),\ r_3 \mapsto (\mathsf{true},\ \{\mathsf{Nurse,General,5}\}),\ \ldots]}$$

Fig. 2. An example of shared resources in a clinic in $\mathcal{R}$PL.

Types T in $\mathcal{R}$PL are basic types B, classes **C**, sets of resource identifiers **Rid** and future types **Fut**$\langle B\rangle$. An asynchronous method invocation is associated to a future variable f of type **Fut**$\langle B\rangle$, where B is the return type of the invoked method. One can see a future as a mailbox that is created by the time a method is asynchronously invoked, and the caller object continues its own execution after the invocation. When the invoked method has completed the execution, the return value will be placed into the mailbox, i.e., the future. Basic types B include integers **Int**, booleans **Bool**, and empty types **Unit**.

Resources R, written as $[r \mapsto (b, \mathcal{Q})]$, is a mapping from resource identifier r to a pair $(b, \mathcal{Q})$, where b is a boolean value indicating the availability of the resource (b evaluates to *true* if the resource is free, and *false* otherwise), and $\mathcal{Q}$ is a set of resource qualities such as category, speciality and efficiency. Fig. 2 shows an example of resources R of a clinic modelled in $\mathcal{R}$PL, where each resource has an identifier with its availability and a set of qualities, e.g., r_1 is a doctor who is available for a new task, and specialises in orthopaedics with three years of experience as qualities.

A class declaration **class C** $\{\overline{T\ x};\ \overline{M}\}$ has a class name **C** and a class body $\{\overline{T\ x};\ \overline{M}\}$ comprising state variables and methods of the class. Methods in $\mathcal{R}$PL have a method signature Sg followed by a method body $\{\overline{T\ x};\ s\}$. A method signature Sg consists of a return type B, method name m and a sequence of formal parameters $\overline{T\ y}$. We assume each method name is unique. We further assume that the formal parameters $\overline{T\ y}$ is a non-empty set and has a fixed pattern **C** $o, \overline{\mathbf{C}'\ o'}, \overline{T'\ x}$ where o is always the callee object identifier of the method of class **C**, $\overline{o'}$ are object identifiers of classes $\overline{\mathbf{C}'}$ and $\overline{x}$ are the remaining parameters.

Statements s, including assignment, conditional, **skip**, **return** and sequential composition, are standard. Statement **wait**(f) suspends the current process until the future variable f is resolved, while other processes in the same object can be scheduled for execution. A future f is resolved when the method associated with f terminates and returns. Statement **cost**(e), the only term in $\mathcal{R}$PL that consumes time, represents e (expression, see below) units of time advancement. Statement **add**(rs) adds new resources to the resource map R, where rs is a set of resource quality set $\mathcal{Q}$, and each of the newly added resources is mapped to a quality set. Statement **release**(rid) frees a set of acquired resources that is indicated by the set of resource identifiers rid.

The right-hand side rhs of an assignment statement includes expressions, object creation, resource acquisition, method invocations and synchronisation. We use the statement **hold**(rs) to acquire resources which have the qualities indicated by rs.

Communication in $\mathcal{R}$PL is based on method calls, which can be either synchronous or asynchronous, respectively written as $m(x,\overline{e})$ **after** $\overline{fs}$ **dl** e' and $!m(x,\overline{e})$ **after** $\overline{fs}$ **dl** e', where x is the callee object and $\overline{e}$ a sequence of formal parameters. In addition, the task dependency of a method call is specified using **after** $\overline{fs}$ in method invocations, where $\overline{fs}$ corresponds to a possibly empty list of the conjunction of future tests $f?$. If the list is empty, $\overline{fs}$ is evaluated to *true*, which means that the method can be invoked without any restriction; otherwise, at least one of the conjunction of future tests in the list must be evaluated to true in order to invoke the method.

Example 3.1. Let $\overline{fs} = fs_1, fs_2$, $fs_1 = f_{11}? \wedge f_{12}?$ and $fs_2 = f_{21}?$. The method call $!m(x,\overline{e})$ **after dl** e' does not have any task dependency, while the call $!m(x,\overline{e})$ **after** $\overline{fs}$ **dl** e' is depending on the two methods associated with f_{11} and f_{12} or on the method associated with f_{21}.

Furthermore, the deadline of finishing a method is specified using **dl** e'. A method without deadline is written as **dl** `null`.

A synchronous method invocation blocks the caller object until the invoked method returns. Asynchronous method invocations, on the contrary, do not block the caller, allowing the caller and callee to run in parallel. Moreover, the synchronisation in $\mathcal{R}$PL is done with f.**get**, which blocks all execution in the object until future f is resolved. The caller object will only be blocked if it tries to retrieve the value of the future with a **get** statement.

Expressions e include constants k, variables x, guards g, self-identifier **this**, `null` expression and a value of resource identifier set rid. A guard g allows a process to release control of an object. It can be boolean conditions b, future test fs, and conjunction of guards.

We assume in this paper that all asynchronously invoked methods inside a conditional statement must be synchronised within the scope of the statement and vice versa. Moreover, we assume that the method invocations inside the body of a conditional statement must not have any dependency on futures associated with asynchronously invoked methods outside the body of a conditional statement and not yet synchronised.

The assumptions described above simplify the realisation of the cost analysis presented in Section 4, which only considers the cost of method invocations that are synchronised, and we aim to implement a type-checker to verify these assumptions in the future.

Let us use a simple example to illustrate the idea of $\mathcal{R}$PL language. Fig. 4 models a simple patient-diagnosis workflow in a clinic, which has the resource map modelled in Fig. 2.

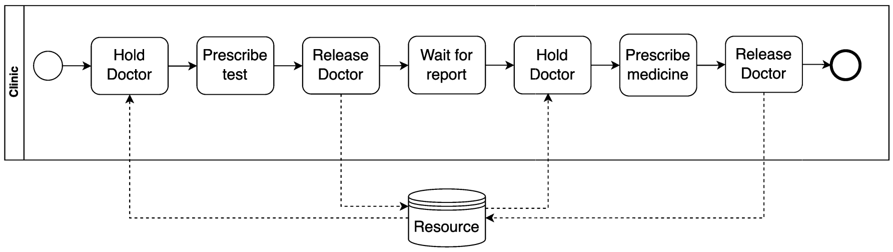

Fig. 3. A simple example.

```
 1  class Clinic{
 2    Pathology lab = new Pathology;
 3    Unit diagnose(Patient p){
 4      Rid r; Fut<Unit> f;
 5      r = hold({Doctor,Ortho,3});
 6      f = !test(lab) after dl null;
 7      release(r);
 8      wait(f);
 9      f.get;
10      r = hold({Doctor,Ortho,3});
11      prescribe(this) after f? dl null; // Write prescription
12      release(r);}
13  }
```

Fig. 4. A simple clinic workflow in $\mathcal{R}$PL.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \varepsilon \mid res \mid fut(f, val) \mid obj(o, a, p, q) & a & ::= & [\ldots, x \mapsto v, \ldots] \\
 & & \mid invoc(o, f, m, \overline{v}, d) \mid error \mid cn\ cn & p & ::= & \texttt{idle} \mid \{l \mid s\} \\
res & ::= & [r \mapsto (b, \mathcal{Q})] & q & ::= & \emptyset \mid \{l \mid s\} \mid q\ q \\
val & ::= & v \mid \bot & s & ::= & \textbf{cont}(f) \mid \textbf{suspend} \mid \ldots \\
v & ::= & o \mid f \mid b \mid k \mid rid \mid \infty & rhs & ::= & m(x, \overline{e})\ \textbf{dl}\ e' \\
 & & & & & \mid !m(x, \overline{e})\ \textbf{dl}\ e' \mid \ldots
\end{array}
$$

Fig. 5. Runtime syntax of $\mathcal{R}$PL.

An orthopaedic doctor (resource) having three years of experience is first acquired on Line 5 before checking the patient p. After checking the patient, the doctor sends a sample to the lab for examination through an asynchronous method invocation on Line 6. While waiting for the lab to send back the result (Line 8), the resources are released such that they can diagnose other patients (Line 7). When the result is ready and retrieved (Line 9), a doctor is acquired again to write the prescription (Lines 10–11), and is released (Line 12) afterwards. For simplicity we do not show the implementation of the method prescribe from Clinic workflow and the Pathology workflow. The corresponding workflow modelled in BPMN is captured in Fig. 3.

3.2. The semantics of $\mathcal{R}$PL

To understand how time advances in $\mathcal{R}$PL and the cost analysis described in Section 4, we briefly discuss the semantics of the language in this section. The semantics of $\mathcal{R}$PL is a transition system whose states are configurations cn described with the runtime syntax defined in Fig. 5.

A configuration cn includes empty configuration ε, resource map res, futures $fut(f, val)$, objects $obj(o, a, p, q)$, message invocations $invoc(o, f, m, \overline{v}, d)$, configuration error $error$ and associative and commutative union operator on configurations (denoted as white space) $cn\ cn$. Resource map res is a mapping from resource identifier r to $(b, \mathcal{Q})$ where b is boolean value and $\mathcal{Q}$ is a set of resource qualities. A future $fut(f, val)$ holds a future identifier f and a return value val, which can be either a value v or $\bot$ indicating that future f has not been resolved. Values v include object identifier o, future identifier f, Boolean values b, Integer or constant values k, resource identifiers set values rid, and `null` expression value ∞.

An object is a term $obj(o, a, p, q)$ where o is the object identifier, a a substitution describing the object's attributes, p an active process, and q a pool of suspended processes. A process p, written as $\{l \mid s\}$, has local variable bindings l and a statement s, or it is `idle`. The pool of suspended processes is indicated by q. We use $q \cup p$ to add a process p to the pool q, and $q \setminus p$ to remove p from the pool. A message invocation is a term $invoc(o, f, m, \overline{v}, d)$, where o is a callee object, f a future to which method m returns a value, $\overline{v}$ the set of actual parameter values and d the deadline for method m.

The statement is extended with **cont**(f) and **suspend**: the former controls the scheduling when a synchronous call completes its execution, returning the control to the caller; and the latter suspends the active process p to the pool of suspended processes q, leaving the processor idle. We extend the right hand side of an assignment with $m(x, \overline{e})$ **dl** e' and

$$\text{(Field-Assign)}\quad \frac{x \in dom(a) \quad v = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} obj(o, a, \{l \mid x = e; s\}, q) \\ \rightarrow obj(o, a[x \mapsto v], \{l \mid s\}, q) \end{array}} \qquad \text{(Local-Assign)}\quad \frac{x \in dom(l) \quad v = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} obj(o, a, \{l \mid x = e; s\}, q) \\ \rightarrow obj(o, a, \{l[x \mapsto v] | s\}, q) \end{array}}$$

$$\text{(Cond-True)}\quad \frac{true = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} obj(o, a, \{l \mid \textbf{if } (e)\ \{s_1\} \textbf{ else } \{s_2\}; s\}, q) \\ \rightarrow obj(o, a, \{l \mid s_1; s\}, q) \end{array}} \qquad \text{(Cond-False)}\quad \frac{false = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} obj(o, a, \{l \mid \textbf{if } (e)\ \{s_1\} \textbf{ else } \{s_2\}; s\}, q) \\ \rightarrow obj(o, a, \{l \mid s_2; s\}, q) \end{array}}$$

$$\text{(Wait-False)}\quad \frac{v = \bot}{\begin{array}{c} obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \\ \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid \textbf{wait}(f); s\})\ fut(f, v) \end{array}} \qquad \text{(Wait-True)}\quad \frac{v \neq \bot}{\begin{array}{c} obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \\ \rightarrow obj(o, a, \{l \mid s\}, q)\ fut(f, v) \end{array}}$$

$$\text{(New-Object)}\quad \frac{o' = \texttt{fresh}() \quad a' = \texttt{atts}(C, o')}{\begin{array}{c} obj(o, a, \{l \mid x = \textbf{new } C; s\}, q) \\ \rightarrow obj(o, a, \{l \mid x = o'; s\}, q)\ obj(o', a', \texttt{idle}, \emptyset) \end{array}} \qquad \text{(Return)}\quad \frac{v = \llbracket e \rrbracket_{(a \circ l)} \quad f = l(destiny)}{\begin{array}{c} obj(o, a, \{l \mid \textbf{return } e; s\}, q)\ fut(f, \bot) \\ \rightarrow obj(o, a, \{l \mid s\}, q)\ fut(f, v) \end{array}}$$

$$\text{(Suspend)}\quad \begin{array}{c} obj(o, a, \{l \mid \textbf{suspend}; s\}, q) \\ \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid s\}) \end{array} \qquad \text{(Skip)}\quad \begin{array}{c} obj(o, a, \{l \mid \textbf{skip}; s\}, q) \\ \rightarrow obj(o, a, \{l \mid s\}, q) \end{array}$$

$$\text{(Activate)}\quad \frac{p = \texttt{select}(q)}{obj(o, a, \texttt{idle}, q) \rightarrow obj(o, a, p, q \setminus p)} \qquad \text{(Context)}\quad \frac{cn = cn'}{cn\ cn'' \rightarrow cn'\ cn''}$$

Fig. 6. Semantics of $\mathcal{R}$PL – part 1.

$!m(x, \overline{e})$ **dl** e', which corresponds to the synchronous and asynchronous method calls at runtime. We use **dl** e' to assign a deadline of e' time units to method m, where $\llbracket e' \rrbracket_{(a \circ l)} > 0$.

The semantics rules of $\mathcal{R}$PL are defined in Figs. 6–9. We use the auxiliary functions $dom(l)$ and $dom(a)$ in the semantics to return the domain of local variables l and object's attributes a, respectively. The evaluation function $\llbracket e \rrbracket_{(a \circ l)}$ returns the value of e by computing the expressions and retrieving the value of identifiers stored either in a or l, where the operator $\circ$ joins the domains of a and l. Moreover, the function $\texttt{atts}(C, o)$ is used to create an object of a class C, which binds $\texttt{this}$ to o, and the function $\texttt{bind}(o, f, m, \overline{v}, d, C)$ returns a process that is going to execute method m with declaration $B\ m(\overline{T\ y})\ \{\overline{T'\ x};\ s\}$, which is defined as:

$$\texttt{bind}(o, f, m, \overline{v}, d, C) = \{[destiny \mapsto f, \overline{y} \mapsto \overline{v}, \overline{x} \mapsto \bot, deadline \mapsto d] \mid s[o/\texttt{this}]\}$$

The semantics rules in Fig. 6 are standard. Rules Field-Assign and Local-Assign assign the value of expression e to an object field and to a local variable, respectively. Rules Cond-True and Cond-False handle conditional statements based on the evaluation of expression e. Rule Wait-False suspends the active process, leaving the object idle if f is not resolved; otherwise, Wait-True consumes **wait**(f). Rule New-Object creates a new object. Rule Return assigns the return value of a method to its future. Rule Skip consumes a **skip** statement in the active process. Rule Suspend moves an active process to the pool of suspended process and the object will become $\texttt{idle}$. If an object is $\texttt{idle}$, the rule Activate $\texttt{selects}$ a suspended process to become active in an idle object, where the $\texttt{select}$ function is defined as follows:

$$\texttt{select}(q) = \begin{cases} \texttt{idle} & \text{if } q = \emptyset \\ p & \text{if } \exists p \in q \text{ and } \texttt{ready}(p) \\ \texttt{idle} & \text{otherwise.} \end{cases} \qquad \texttt{ready}(p) = \begin{cases} \text{true} & \text{if } p = \textbf{wait}(f), \text{ where} \\ & \quad fut(f, v) \text{ and } v \neq \bot \\ \text{false} & \text{otherwise.} \end{cases}$$

Fig. 7 captures the communications between objects in $\mathcal{R}$PL. Rules Sync-Call and Async-Call handle the communication between objects through method invocations. These two rules rewrite method invocations to a conditional statement to ensure that the task dependencies between method calls have been fulfilled. If at least one of the conjunction of future tests in $\overline{fs}$ is evaluated to true, method m is invoked synchronously by rule Sync-Run or Self-Sync-Run (or asynchronously by rule Async-Run); otherwise, the process will be suspended.

Example 3.2. Let $fs_1 = f_{11}? \wedge f_{12}?$, $fs_2 = f_{21}?$ and $fs_3 = f_{31}$. Assume the future test $f_{21}?$ is *true*, while the others are *false*. For the method call $!m(x, \overline{e})$ **after** fs_1, fs_2 **dl** e', the boolean condition fs_1, fs_2 will be evaluated to *true*, and it will be written to $!m(x, \overline{e})$ **dl** e'. Whereas for the method call $!m(x, \overline{e})$ **after** fs_1, fs_3 **dl** e', the boolean condition fs_1, fs_2 will be evaluated to *false*, and the process will be suspended.

$$\begin{array}{c}(\textsc{Sync-Call})\\ obj(o,a,\{l \mid x = m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\},q)\\ \rightarrow obj(o,a,\{l \mid \textbf{if}\ (\overline{fs})\ \{x = m(x',\overline{e})\ \textbf{dl}\ e';s\}\ \textbf{else}\ \{\textbf{suspend}; x = m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\},q)\end{array}$$

$$(\textsc{Self-Sync-Run})\quad \frac{\begin{array}{c} o = x' \quad \overline{v} = [\![\overline{e}]\!]_{(a\circ l)} \quad f = l(destiny) \quad d = [\![e']\!]_{(a\circ l)}\\ f' = \texttt{fresh}() \quad \{l' \mid s'\} = \texttt{bind}(o,f',m,\overline{v},d,\texttt{class}(o))\end{array}}{\begin{array}{c} obj(o,a,\{l \mid x = m(x',\overline{e})\ \textbf{dl}\ e';s\},q)\\ \rightarrow obj(o,a,\{l' \mid s';\textbf{cont}(f)\},q \cup \{l \mid x = f'.\textbf{get};s\})\ fut(f',\bot)\end{array}}$$

$$(\textsc{Sync-Run})\quad \frac{o' = x' \quad o \neq o' \quad f = \texttt{fresh}()}{\begin{array}{c} obj(o,a,\{l \mid x = m(x',\overline{e})\ \textbf{dl}\ e';s\},q)\ obj(o',a',p,q')\\ \rightarrow obj(o,a,\{l \mid f = !m(x',\overline{e})\ \textbf{dl}\ e'; x = f.\textbf{get};s\},q)\ obj(o',a',p,q')\end{array}}$$

$$\begin{array}{c}(\textsc{Async-Call})\\ obj(o,a,\{l \mid x = !m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\},q)\\ \rightarrow obj(o,a,\{l \mid \textbf{if}\ (\overline{fs})\ \{x = !m(x',\overline{e})\ \textbf{dl}\ e';s\}\ \textbf{else}\ \{\textbf{suspend}; x = !m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\},q)\end{array}$$

$$(\textsc{Async-Run})\quad \frac{\overline{v} = [\![\overline{e}]\!]_{(a\circ l)} \quad o' = x' \quad f = \texttt{fresh}() \quad d = [\![e']\!]_{(a\circ l)}}{\begin{array}{c} obj(o,a,\{l \mid x = !m(x',\overline{e})\ \textbf{dl}\ e';s\},q)\\ \rightarrow obj(o,a,\{l \mid x = f;s\},q)\ invoc(o',f,m,\overline{v},d)\ fut(f,\bot)\end{array}}$$

$$(\textsc{Invoc})\quad \frac{\{l \mid s\} = \texttt{bind}(o,f,m,\overline{v},d,\texttt{class}(o))}{obj(o,a,p,q)\ invoc(o,f,m,\overline{v},d) \rightarrow obj(o,a,p,q \cup \{l \mid s\})}$$

$$(\textsc{Sync-Return-Sched})\quad \frac{f = l(destiny)}{\begin{array}{c} obj(o,a,\{l' \mid \textbf{cont}(f)\}),q \cup \{l \mid s\}\\ \rightarrow obj(o,a,\{l \mid s\},q)\end{array}} \qquad (\textsc{Get})\quad \frac{v \neq \bot}{\begin{array}{c} obj(o,a,\{l \mid x = f.\textbf{get};s\},q)\ fut(f,v)\\ \rightarrow obj(o,a,\{l \mid x = v;s\},q)\ fut(f,v)\end{array}}$$

Fig. 7. Semantics of $\mathcal{R}$PL – part 2.

$$(\textsc{Add-Resource-1})\quad \frac{\begin{array}{c} rs \neq \emptyset \quad rs = \mathcal{Q} \cup rs' \quad r = \texttt{fresh}()\\ res' = res[r \mapsto (true,\mathcal{Q})]\end{array}}{\begin{array}{c} obj(o,a,\{l \mid \textbf{add}(rs);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid \textbf{add}(rs');s\},q)\ res'\end{array}} \qquad (\textsc{Release-Resource-1})\quad \frac{\begin{array}{c} rid = \{r\} \cup rid' \quad res(r) = (false,\mathcal{Q})\\ res' = res[r \mapsto (true,\mathcal{Q})]\end{array}}{\begin{array}{c} obj(o,a,\{l \mid \textbf{release}(rid);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid \textbf{release}(rid');s\},q)\ res'\end{array}}$$

$$(\textsc{Add-Resource-2})\quad \frac{rs = \emptyset}{\begin{array}{c} obj(o,a,\{l \mid \textbf{add}(rs);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid s\},q)\ res\end{array}} \qquad (\textsc{Release-Resource-2})\quad \frac{rid = \emptyset}{\begin{array}{c} obj(o,a,\{l \mid \textbf{release}(rid);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid s\},q)\ res\end{array}}$$

$$(\textsc{Hold-Resource-1})\quad \frac{rs \neq \emptyset \quad (res',rid) = \texttt{holdRes}(rs,res,\emptyset) \quad rid \neq \emptyset}{\begin{array}{c} obj(o,a,\{l \mid x = \textbf{hold}(rs);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid x = rid;s\},q)\ res'\end{array}} \qquad (\textsc{Hold-Resource-2})\quad \frac{rs = \emptyset}{\begin{array}{c} obj(o,a,\{l \mid x = \textbf{hold}(rs);s\},q)\ res\\ \rightarrow obj(o,a,\{l \mid s\},q)\ res\end{array}}$$

Fig. 8. Semantics of $\mathcal{R}$PL – part 3.

Rule Self-Sync-Run directly transfers the control of the object from the caller to the callee. After the execution of invoked method is completed, rule Sync-Return-Sched reactivates the caller. Rule Sync-Run specifies a synchronous call to another object, which is replaced by an asynchronous call followed by a **get** statement. Rule Async-Run creates an invocation message to o' with a fresh unresolved future f, method name m, actual parameters $\overline{v}$ and deadline d. Rule Invoc adds a process (that is going to execute the method) to the pool of suspended processes. Rule Get retrieves the value of future f if it is resolved; the reduction on this object is blocked otherwise.

Resources are handled by the semantics rules in Fig. 8. The two Add-Resource rules recursively add new resources r to the resource map res, based on rs that is a set of resource quality set $\mathcal{Q}$. Each newly added resource r has a set of quality $\mathcal{Q} \in rs$, which is removed from rs when r is added to the map res. The two rules Release-Resource recursively return the acquired resources rid. The resource acquisition is handled by the two Hold-Resource rules (see also the function `holdRes` defined below). Each of the acquired resources has a quality set $\mathcal{Q} \in rs$. Note that it is required to have all the requested resources to be available in order to consume the **hold** statement.

$$\frac{\llbracket e \rrbracket_{(a \circ l)} = 0}{obj(o, a, \{l \mid \textbf{cost}(e); s\}, q) \to obj(o, a, \{l \mid s\}, q)}\ (\textsc{Cost}) \qquad \frac{\begin{array}{c}\texttt{strongstable}_{\texttt{t}}(cn)\\ cn' = \texttt{checkDl}(cn, t)\\ error \notin cn'\end{array}}{cn \to \texttt{advance}(cn', t)}\ (\textsc{Tick-Ok}) \qquad \frac{\begin{array}{c}\texttt{strongstable}_{\texttt{t}}(cn)\\ cn' = \texttt{checkDl}(cn, t)\\ error \in cn'\end{array}}{cn \to error}\ (\textsc{Tick-Miss})$$

Fig. 9. Semantics of $\mathcal{R}$PL – part 4.

$$\texttt{holdRes}(rs, res, rid) = \begin{cases} \texttt{holdRes}(rs', res[r \mapsto (false, \mathcal{Q})], \{r\} \cup rid) & \text{if } rs = \{\mathcal{Q}\} \cup rs', \text{and} \\ & \exists r.res(r) = (true, \mathcal{Q}) \\ (res, rid) & \text{if } rs = \emptyset \\ (res, \emptyset) & \text{otherwise.} \end{cases}$$

Fig. 9 takes care of the time advancement in the language. In $\mathcal{R}$PL, the unique statement that consumes time is **cost**(e). Rule Cost specifies a trivial case when e evaluates to 0. When the configuration cn reaches a *stable* state, i.e., no other transition is possible except those evaluating the **cost**(e) statement where e is evaluated to some $t \geq 0$, time advances by the smallest value required to let at least one process execute. To formalize this semantics, we have defined stability below:

Definition 3.1. A configuration is t-stable for some $t > 0$, denoted as $\texttt{stable}_{\texttt{t}}(cn)$, if every object in cn is in one of the following forms:

1. $obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)$ where $fut(f, \bot) \in cn$,
2. $obj(o, a, \{l \mid \textbf{cost}(e); s\}, q)$ where $\llbracket e \rrbracket_{(a \circ l)} \geq t$,
3. $obj(o, a, \{l \mid \textbf{hold}(rs); s\}, q)$ where $res \in cn$ and $\exists \mathcal{Q} \in rs$ s.t. $\nexists r \in res$, $res(r) = (true, \mathcal{Q})$,
4. $obj(o, a, \texttt{idle}, q)$ and if
   (a) $q = \emptyset$, or,
   (b) $\forall p \in q$ and if
      i. $p = \{l \mid \textbf{wait}(f); s\}$ and $fut(f, \bot) \in cn$, or,
      ii. $p = \{l \mid x = m(x', \overline{e})$ **after** $\overline{fs}$ **dl** $e'; s\}$, or
      $p = \{l \mid x = !m(x', \overline{e})$ **after** $\overline{fs}$ **dl** $e'; s\}$, where $\overline{fs} = false$.

A configuration cn is *strongly t-stable*, written as $\texttt{strongstable}_{\texttt{t}}(cn)$, if it is t-stable and there is an object $obj(o, a, \{l \mid \textbf{cost}(e); s\}, q)$ with $\llbracket e \rrbracket_{(a \circ l)} = t$. Note that both t-stable and strongly t-stable configurations cannot proceed any more because every object is stuck either on a **cost**(e), on unresolved futures, or waiting for some resources. Rules Tick-Ok and Tick-Miss advance time when the configuration cn is strongly t-stable. We first use the function $\texttt{checkDl}(cn, t)$ to check if any invoked methods will violate their own deadline if time advances by t units, which is defined as follows, where dl is the shorthand for *deadline*:

$$\texttt{checkDl}(cn, t) = \begin{cases} obj(o, a, \{l[dl \mapsto t'] \mid s\}, q')\ \texttt{checkDl}(cn', t) & \text{if } cn = obj(o, a, \{l \mid s\}, q)\ cn', \text{and} \\ & l(dl) - t = t' \geq 0, \text{and} \\ & \forall l'.\{l' \mid \_\_\} \in q\ \wedge\ l'(dl) - t \geq 0, \text{and} \\ & q' = \texttt{updateDl}(q, t) \\ error & \text{if } cn = obj(o, a, \{l \mid s\}, q)\ cn', \text{and} \\ & l(dl) - t < 0, \text{or} \\ & \exists l'.\{l' \mid \_\_\} \in q\ st.\ l'(dl) - t < 0 \\ cn & \text{otherwise,} \end{cases}$$

and $\texttt{updateDl}(q, t) = \begin{cases} \{l[dl \mapsto l(dl) - t] \mid s\} \cup \texttt{updateDl}(q', t) & \text{if } q = \{l \mid s\} \cup q' \\ \emptyset & \text{if } q = \emptyset\ . \end{cases}$

Rule Tick-Ok corresponds to the case where none of the invoked methods violates its own deadline and time advances for the whole configuration as defined below:

$$\texttt{advance}(cn, t) = \begin{cases} obj(o, a, \{l \mid \textbf{cost}(k); s\}, q)\ \texttt{advance}(cn', t) & \text{if } cn = obj(o, a, \{l \mid \textbf{cost}(e); s\}, q)\ cn', \text{and} \\ & k = \llbracket e \rrbracket_{(a \circ l)} - t \\ obj(o, a, \{l \mid \textbf{hold}(u); s\}, q)\ \texttt{advance}(cn', t) & \text{if } cn = obj(o, a, \{l \mid \textbf{hold}(u); s\}, q)\ cn' \\ obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)\ \texttt{advance}(cn', t) & \text{if } cn = obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)\ cn' \\ obj(o, a, \texttt{idle}, q)\ \texttt{advance}(cn', t) & \text{if } cn = obj(o, a, \texttt{idle}, q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

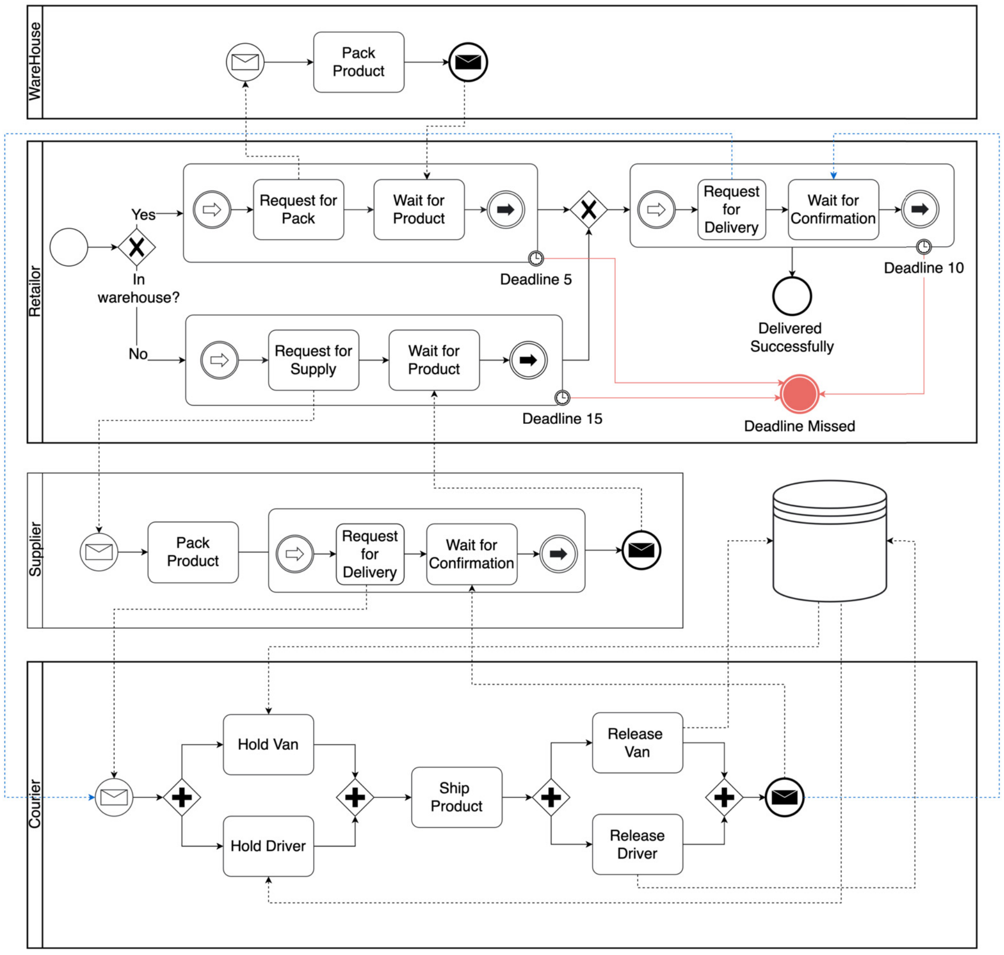

Fig. 10. An illustrative example.

If there exists a method that violates its own deadline, the configuration is rewritten to an error state and cannot proceed any further (see rule TICK-MISS).

The *initial configuration* of an $\mathcal{R}$PL program with main method $\{\overline{T\ x};\ s\}$ is

$$obj(o_{main}, \varepsilon, \{[destiny \mapsto f_{initial}, \overline{x} \mapsto \bot, deadline \mapsto \infty\}, q)$$

where o_{main} is object name, and $f_{initial}$ is a fresh future name. Normally, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$ and $\overset{t}{\Rightarrow}$ is $\rightarrow^* \overset{t}{\rightarrow} \rightarrow^*$. A computation is $cn \overset{t_1}{\Rightarrow} \ldots \overset{t_n}{\Rightarrow} cn'$; that is, cn' is a configuration reachable from cn with either transitions $\rightarrow$ or $\overset{t}{\Rightarrow}$. When the time labels of transitions are not necessary, we also write $cn \Rightarrow^* cn'$.

Definition 3.2. The computational time of $cn \overset{t_1}{\Rightarrow} \ldots \overset{t_n}{\Rightarrow} cn'$ is $t_1 + \cdots + t_n$.

The computational time of a configuration cn, written as `time`(cn), is the maximum computational time of computations starting at cn. The computational time of an $\mathcal{R}$PL program is the computational time of its initial configuration.

3.3. An illustrative example of modelling cross-organisational workflows in $\mathcal{R}$PL

In this section, we present a simple motivating example to explain how $\mathcal{R}$PL can be employed in modelling cross-organisational workflows. Fig. 10 depicts a cross-organisational workflow of order fulfilment in BPMN. Fig. 11 shows the corresponding workflows modelled in $\mathcal{R}$PL. The code snippet captures the collaboration between the workflows of a retailer, a supplier and a courier company. Line 1 models the available resources. Lines 3–13 define a retail-sale workflow. Line 4 declares local variables. If the product is available in the warehouse, a request to the warehouse to pack the product is made asynchronously with associated future f_1 and a deadline of 5 times unit on Line 7 (without any task dependency).

```
1  [r1 ↦(true,{Driver,TruckDriver,5}),r2 ↦(true,{Driver,VanDriver,5}),r3↦(true,{Van,Delivery,1500}),. . .]
```

```
2  class Retailer{
3    Unit sale(Retailer rt, Supplier sp, Courier cr){
4      Fut<Unit> f1; Fut<Unit> f2; Fut<Unit> f3; Warehouse wh;
5      wh = new Warehouse;
6      if(In−Warehouse){
7        f1 = !pack(wh) after dl 5;
8        wait(f1);} // wait for packing
9      else{
10       f2 = !supply(sp,cr) after dl 15;
11       wait(f2);} // wait for supply
12     f3 = !deliver(cr) after dl 10;
13     wait(f3);} // wait for delivery confirmation
14 }
```

```
15 class Warehouse{
16   Unit pack(Warehouse wh){
17     cost(k1);} // Packing time
18 }
```

```
19 class Supplier{
20   Unit supply(Supplier sp, Courier cr){
21     Fut<Unit> f4;
22     cost(k2); // Packing time
23     f4 = !deliver(cr) after dl 10;
24     wait(f4);}
25 }
```

```
26 class Courier{
27   Unit deliver(Courier cr){
28     Rid r;
29     r = hold({Driver,VanDriver,5},{Van,Delivery,1500});
30     cost(k3); // Delivery Time
31     release(r);}
32 }
```

```
33 {
34   Retailer rt; Supplier sp; Courier cr;
35   rt = new Retailer; sp = new Supplier; cr = new Courier;
36   sale(rt,sp,cr) after dl null;
37 }
```

Fig. 11. An example of collaborative workflows in $\mathcal{R}$PL.

Otherwise, a request to the supplier to supply the product is made asynchronously with associated future f_2 and a deadline of 15 times unit on Line 10. While waiting for the product (either on Line 8 or Line 11), the retailer can continue with other tasks. After getting the product either from the warehouse (f_1 is resolved) or from the supplier (f_2 is resolved), it is sent to the customer by utilising the services of a courier company (Line 12) with a deadline of 10 times unit. While waiting for the confirmation of delivery (until f_3 is resolved), the retailer can again continue with other tasks.

Lines 16–17 define pack workflow of the warehouse, where Line 17 models the packing time. Lines 20–24 define supply workflow of the supplier. After packing the product, the supplier sends a request asynchronously to the courier to deliver the product to the retailer on (Line 23).

Lines 27–31 define the deliver workflow of the courier company. A driver and a van (resources with some specific qualities) are first acquired to deliver the product (Line 29). Line 30 depicts the time taken for delivery. Afterwards, the acquired resources are released (Line 31).

Lines 33–37 define the main method body, where on Line 36 the sale workflow of the retailer is called synchronously and objects of supplier and courier are also passed as arguments to allow the collaboration between their workflows.

4. Cost analysis of $\mathcal{R}$PL program

In this section, we describe the cost analysis for workflows modelled in $\mathcal{R}$PL. The analysis translates an $\mathcal{R}$PL program into a set of cost equations that can be fed to a constraint solver. The solution to the resulting constraint set is an over-approximation of the execution time of the $\mathcal{R}$PL program. We use the example in Fig. 11 to illustrate the idea of the analysis. The analysis assumes all $\mathcal{R}$PL programs terminate and are well-typed, and all invoked methods are synchronised. It extends the analysis presented in [31] by handling a more expressive language with explicit notions of task dependencies, resource allocations and two-way selection.

A cost equation in the analysis results in a cost expression *exp* that is defined as follows:

$$exp ::= k \mid c_m \mid max(exp, exp) \mid exp + exp$$

A cost expression may have natural numbers k, the cost c_m of executing a method m, the maximum and the sum of two cost expressions.

Given an $\mathcal{R}$PL program $\mathcal{P}$, the analysis iterates over every method definition $B\ m(\overline{T\ y})\{\overline{T\ x};\ s\}$ in each class in $\mathcal{P}$, and translates it into a cost equation of the form $eq_m{=}exp$, where *exp* corresponds to an upper bound of the computational time of m. The analysis performs this translation by considering the process pool of every object associated with the execution of method m, computing an upper bound of the finishing time of all of its processes, which gives rise to an upper bound of the computational time of the method itself.

In the following, we describe the two significant structures, namely, *synchronisation schema* and *accumulated costs*, used in the analysis to handle the complexity of considering process pools.

4.1. Synchronisation schema

We will first describe synchronisation sets, an element of synchronisation schema, and proceed with the function that is used to manipulate the schema. A synchronisation set [31], ranged over $O, O', \ldots,$ is a set of object identifiers whose processes have implicit dependencies; that is, the processes of these objects may reciprocally influence the process pools of the other objects in the same set through method invocations and synchronisations.

A *synchronisation schema*, ranged over $S, S', \ldots,$ is a set of pairwise disjoint synchronisation sets. Let $B\ m(C\ o, \overline{C'\ o'}, \overline{T\ x})\ \{\overline{T'\ x'};\ s\}$ be an $\mathcal{R}$PL method declaration. The synchronisation schema of m, denoted as S_m, can be seen as a distribution of the objects used in that method into synchronisation sets, where $S_m = \mathtt{sschem}(\{\{o, \overline{o'}\}\}, s, o)$, which is defined in Definition 4.1.

Definition 4.1 *(Synchronisation Schema Function).* Let S be a synchronisation schema, s a statement and o a carrier object which is executing s.

$$\mathtt{sschem}(S, s, o) = \begin{cases} S \oplus \{o', \overline{o''}\} & \text{if } s \text{ is } x = m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{fs} \textbf{ dl } e' \\ & \quad \text{or, } x = !m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{fs} \textbf{ dl } e' \\ \mathtt{sschem}(\mathtt{sschem}(S, s', o), s'', o) & \text{if } s \text{ is } \textbf{if } (e)\ \{s'\} \textbf{ else } \{s''\} \\ \mathtt{sschem}(\mathtt{sschem}(S, s', o), s'', o) & \text{if } s \text{ is } s'; s'' \\ S & \text{otherwise,} \end{cases}$$

where

$$S \oplus O = \begin{cases} \{O\} & \text{if } S = \emptyset \\ (S' \oplus O) \cup \{O'\} & \text{if } S = S' \cup \{O'\} \text{ and } O' \cap O = \emptyset \\ S' \oplus (O' \cup O) & \text{if } S = S' \cup \{O'\} \text{ and } O' \cap O \neq \emptyset \end{cases}$$

The term $S(o)$ represents the synchronisation set containing o in the synchronisation schema S. The function $S \oplus O$ merges a schema S with a synchronisation set O. If none of the objects in O belongs to a set in S, the function reduces to a simple set union. For example, let $S = \{\{o_1, o_2\}, \{o_3, o_4\}\}$. Then $S \oplus \{o_2, o_5\}$ is equal to $(\{\{o_1, o_2\}\} \oplus \{o_2, o_5\}) \cup \{\{o_3, o_4\}\}$, resulting $\{\{o_1, o_2, o_5\}, \{o_3, o_4\}\}$.
To perform cost analysis later, a synchronisation schema will be constructed for each method m. The synchronisation schemas of the methods defined in Fig. 11 are $S_{sale} = \{\{rt, sp, cr\}, \{wh\}\}$, $S_{pack} = \{\{wh\}\}$, $S_{supply} = \{\{sp, cr\}\}$, $S_{deliver} = \{\{cr\}\}$, $S_{main} = \{\{o_{main}\}, \{rt, sp, cr\}\}$.

4.2. Accumulated costs

The syntax of *exp* is extended to express (an over-approximation of) the time progressions of processes in the same synchronisation set. We call this extension *accumulated cost* [31], denoted as $\mathcal{E}$, which is defined as follows:

$$\mathcal{E} ::= exp \mid \mathcal{E} \cdot \langle c_m, exp \rangle \mid \mathcal{E} \parallel exp$$

Let o be a carrier object and o' an object that does not belong to the same synchronisation set of o, i.e., $o' \notin S(o)$, for a given synchronisation schema S. The term *exp* represents the starting time of a process running on o'. The term $\mathcal{E} \cdot \langle c_m, exp \rangle$

describes the starting time of a method invoked asynchronously on object o'. For example, when o invokes a method m on o' using $f = !m(o', \overline{o''}, \overline{e})$ **after** $\overline{fs}$ **dl** e', the accumulated cost of the synchronisation set of o' is $\mathcal{E} \cdot \langle c_m, 0\rangle$, where $\mathcal{E}$ is the cost accumulated up to that point where the method m is invoked, and c_m is the cost of executing method m. Statement **cost**(e) in the process of the carrier o not only advances time in o, but also updates the starting time of succeeding method invocations on object o' to $\mathcal{E} \cdot \langle c_m, e\rangle$, indicating that the starting time of the subsequent method invocation on the synchronisation set of o' is after the time expressed by $\mathcal{E}$ plus the maximum between c_m and e. The term $\mathcal{E} \parallel exp$ expresses the time advancement in the carrier object o when a method running in parallel on an object o' in another synchronisation set is synchronised. In this situation, the time advances by the maximum between the current time exp in o and the time $\mathcal{E}$ in o'. The evaluation function for the accumulated cost, denoted as $[\![\mathcal{E}]\!]$, computes the starting time of the next process in the synchronisation set whose cost is $\mathcal{E}$ as follows:

$$[\![exp]\!] = exp, \quad [\![\mathcal{E} \cdot \langle c_m, exp\rangle]\!] = [\![\mathcal{E}]\!] + max(c_m, exp), \quad [\![\mathcal{E} \parallel exp]\!] = max([\![\mathcal{E}]\!], exp)$$

The table below shows the accumulated costs of some of the statements declared in Fig. 11.

Method	*Line*	*Accumulated Cost*
m_{sale}	7	$0 \cdot \langle c_{pack}, 0\rangle$
m_{sale}	10	$0 \cdot \langle c_{pack}, 0\rangle \parallel 0 \cdot \langle c_{supply}, 0\rangle$
m_{sale}	12	$(0 \cdot \langle c_{pack}, 0\rangle \parallel 0 \cdot \langle c_{supply}, 0\rangle) \cdot \langle c_{deliver}, 0\rangle$
m_{pack}	17	k_1
m_{supply}	22	k_2
m_{supply}	23	$k_2 \cdot \langle c_{deliver}, 0\rangle$
$m_{deliver}$	30	k_3

4.3. Translation function

This section defines the translation function that computes the cost of a method by analysing all possible synchronisation sets and synchronisations made on it. Given an $\mathcal{R}$PL method m and a synchronisation schema S_m computed based on Section 4.1, the translate function analyses the body of the method m by parsing each of its statements sequentially and recording the accumulated costs of synchronisation sets in a translation environment.

Given a synchronisation schema of a method m, S_m, the translation function $\mathcal{T}_{S_m}(I, \Psi, o, t_a, t, s)$ defined in Fig. 12 takes six parameters: I is a map from future names to synchronisation sets, Ψ a translation environment, o is the carrier object, t_a a cost expression that computes the cost of the methods invoked on objects belonging to the same synchronisation set of carrier o but not yet synchronised, t a cost expression that computes the computational time accumulated from the start of the method execution, and a statement s.

The function returns a tuple of four elements: an updated map I', an updated translation environment Ψ', the updated cost of asynchronously running objects t'_a, and the updated current cost t'.

Definition 4.2 *(Translation Environment).* Translation environments, ranged over $\Psi, \Psi', \ldots$, is a mapping from synchronisation sets to their corresponding accumulated costs ($S_m(o) \mapsto \mathcal{E}$).

We explain in the following each of the cases of the $\mathcal{T}$ function defined in Fig. 12.
Case 1: Each statement in a sequential composition is translated recursively.
Case 2: When s is a **cost**(e) statement, the function updates the current cost t and the accumulated cost Ψ by adding the cost e to them.
Case 3: When s is a $m'(o', \overline{e})$ **after** $\overline{fs}$ **dl** e' (synchronous method invocations), the method can only be invoked if the futures it depends on are resolved. We need to first compute the cost of all methods associated with futures belonging to $\overline{fs}$ using the function `getF`, which extracts the future identifiers from a conjunction of future tests fs. Then, for each $fs \in \overline{fs}$, we use the **trans** function defined in Fig. 13 (see below for explanation) to compute a tuple with elements including the cost of asynchronously running objects t'_a and the corresponding current cost t' of all the methods associating to the futures belonging to fs. Afterwards, we take the maximum mt_a of all the computed t'_a as the resulting updated cost of objects asynchronously running in parallel with the carrier o. Similarly, we take the maximum mt of the all computed t' as the cost of executing the methods associating $\overline{fs}$, and add the cost of method m', $c_{m'}$ to mt and Ψ. Note that the functions $t_a(\mathcal{D})$ and $t(\mathcal{D})$ extract the elements t'_a and t', respectively, from all the tuples in $\mathcal{D}$.
Case 4 & 5: The next two cases correspond to s as an asynchronous method invocation $!m'(o', \overline{e})$ **after** $\overline{fs}$ **dl** e'. Similar to **Case 3**, we first compute the cost of all methods associating with futures belonging to $\overline{fs}$. **Case 4** handles the situation if carrier o and callee o' are in the same synchronisation set. We add the cost of method m to mt_a and update I with the binding $f \mapsto S_m(x)$. If o' is not in the same synchronisation set of carrier o, as in **Case 5**, we add the binding $f \mapsto S_m(y)$ to I and update the Ψ by adding the cost of method m' to the accumulated cost of $S_m(y)$.
Case 6: When s is either f.**get** or **wait**(f) statement, we compute the cost by utilising function $\textbf{trans}_{S_m}(I, \Psi, x, t_a, t, \{f\})$.

$$
\mathcal{T}_{S_m}(I,\Psi,o,t_a,t,s) =
\begin{cases}
1.\quad \mathcal{T}_{S_m}(I',\Psi',o,t'_a,t',s'') & \text{if } s \text{ is } s';s'', \text{ and} \\
 & \quad (I',\Psi',t'_a,t') = \mathcal{T}_{S_m}(I,\Psi,o,t_a,t,s') \\
2.\quad (I,\Psi+e,t_a,t+e) & \text{if } s \text{ is } \textbf{cost}(e) \\
3.\quad (I,\Psi+c_{m'},mt_a,mt+c_{m'}) & \text{if } s \text{ is } x = m'(o',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e',\ \text{ and} \\
 & \mathcal{D} = \{(I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\texttt{getF}(fs)) \mid fs \in \overline{fs}\}, \\
 & \text{and,} \\
 & mt_a = max(t_a(\mathcal{D})), \text{ and} \\
 & mt = max(t(\mathcal{D})) \\
4.\quad (I[f \mapsto S_m(o)],\Psi,mt_a+c_{m'},mt) & \\
 & \text{if } s \text{ is } f = !m'(o',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e',\ \text{ and } o' \in S_m(o), \text{ and} \\
 & \mathcal{D} = \{(I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\texttt{getF}(fs)) \mid fs \in \overline{fs}\}, \\
 & \text{and,} \\
 & mt_a = max(t_a(\mathcal{D})), \text{ and} \\
 & mt = max(t(\mathcal{D})) \\
5.\quad (I[f \mapsto S_m(o')],\Psi[S_m(o') \mapsto \mathcal{E}\cdot\langle c_{m'},0\rangle],mt_a,mt) & \\
 & \text{if } s \text{ is } f = !m'(o',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e',\ \text{ and } o' \notin S_m(o), \text{ and} \\
 & \mathcal{D} = \{(I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\texttt{getF}(fs)) \mid fs \in \overline{fs}\}, \\
 & \text{and,} \\
 & mt_a = max(t_a(\mathcal{D})), \text{ and} \\
 & mt = max(t(\mathcal{D})) \\
 & \text{where} \\
 & \quad \mathcal{E} = \begin{cases} \Psi(S_m(o')) & \text{if } S_m(o') \in dom(\Psi) \\ mt & \text{otherwise.} \end{cases} \\
6.\quad (I',\Psi',t'_a,t') & \text{if } s \text{ is } f.\textbf{get} \text{ or } \textbf{wait}(f),\ \text{ and} \\
 & \quad (I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\{f\}) \\
7.\quad (I,\Psi,max(t'_a,t''_a),max(t',t'')) & \\
 & \text{if } s \text{ is } \textbf{if}\ (e)\ \{s\}\ \textbf{else}\ \{s'\}, \text{ and} \\
 & \quad (I',\Psi',t'_a,t') = \mathcal{T}_{S_m}(I,\Psi,o,t_a,t,s) \\
 & \quad (I'',\Psi'',t''_a,t'') = \mathcal{T}_{S_m}(I,\Psi,o,t_a,t,s') \\
8.\quad (I,\Psi,t_a,t) & \text{otherwise.}
\end{cases}
$$

Fig. 12. The translation function.

$$
\textbf{trans}_{S_m}(I,\Psi,o,t_a,t,F) =
\begin{cases}
(a)\ \ (I,\Psi,t_a,t) & \text{if } F = \emptyset \\
(b)\ \ \textbf{trans}_{S_m}(I \setminus F'',\Psi+t_a,o,0,t+t_a,F') & \text{if } F = F' \cup \{f\} \text{ and } o \in I(f) \text{ and} \\
 & \ F'' = \{f' \mid I(f') = S_m(o)\} \\
 & \\
(c)\ \ \textbf{trans}_{S_m}(I \setminus F'',(\Psi \parallel t') \setminus I(f),o,0,t',F') & \\
 & \text{if } F = F' \cup \{f\} \text{ and } o \notin I(f) \text{ where} \\
 & \ F'' = \{f' \mid I(f') = S_m(o) \vee I(f') = I(f)\} \\
 & \ \text{and } t' = max(t+t_a, [\![\Psi(I(f))]\!]) \\
 & \\
(d)\ \ \textbf{trans}_{S_m}(I \setminus F'',\Psi+t_a,o,0,t+t_a,F') & \text{if } F = F' \cup \{f\} \text{ and } f \notin dom(I) \text{ where} \\
 & F'' = \{f' \mid I(f') = S_m(o)\}
\end{cases}
$$

Fig. 13. The auxiliary translation function.

Case 7: To handle conditional statements, we first calculate the cost of executing the statements in the conditional branch. Since the conditional branch may be executed at runtime, to over-approximate the cost, we update t_a with the maximum of t'_a and t''_a, and the current cost t with the maximum of t' and t''. As we have assumed the all methods invoked in a conditional branch will be synchronised within the same branch, we do not change I and Ψ.

The trans function.

Similar to the translation function $\mathcal{T}$, the auxiliary function **trans** in Fig. 13 also takes six arguments. The auxiliary function **trans** in Fig. 13 also takes six arguments.

While the first five are the same as those of $\mathcal{T}$, the last one is a set of futures F. This function recursively calculates the cost of each method associated to the futures in F as follows:

(a): It is trivial if F is an empty set, where I, Ψ, t_a, and t remain unchanged.

(b): This corresponds to the case where F contains a future f associated to a method call whose callee belongs to same synchronisation set of the carrier x. Since it is non-deterministic when this method will be scheduled for execution, to over-approximate the cost, we sum the cost of the methods invoked on the objects that are in $S_m(o)$, which is stored in t_a, and add it to the cost t accumulated so far. We then reset t_a to 0 and remove all the corresponding futures from I since the related costs have been already considered.

(c): When F contains a future associated to a method call whose callee (say o') does not belong to $S_m(o)$. Since objects o

and o' reside in separate synchronisation sets, the method running on o' runs in parallel with o. Therefore, the cost is the maximum between the total cost of all methods invoked on the objects in $S_m(o)$ and that in $S_m(o')$. Since we over-approximating the cost, the cost of all methods invoked on the objects in $S_m(o)$ and $S_m(o')$ have already been computed. Therefore, we remove $S_m(o')$ from Ψ, as well as all the futures associated with $S_m(o)$ and $S_m(o')$ from I.
(d): When F contains a future f that does not belong to I, it indicates that the cost of the method corresponding to f has been already calculated. Since it can happen that other methods may be invoked after this computation, the actual termination of the method invocation corresponding to f may happen after the completion of these invocations. To take this into account, we add the cost of all methods whose callee belongs to $S_m(o)$, which has been stored in t_a, to the cost accumulated so far.

Example 4.1. We show how the translation function can be applied on the methods defined in Fig. 11.

Let $S_{sale} = \{\{rt, sp, cr\}, \{wh\}\}$, $S_{pack} = \{\{wh\}\}$, $S_{supply} = \{\{sp, cr\}\}$, $S_{deliver} = \{\{cr\}\}$ and $S_{main} = \{\{o_{main}\}, \{rt, sp, cr\}\}$ (as computed in Section 4.1). We use s_i to indicate the sequence of statements of a method body starting from line i.

Translation of method *sale* :
$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, if\,(In - Warehouse)\{s_7; s_8\}\ else\ \{s_{10}; s_{11}\}\ s_{12})$
$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, f_1 = !pack(wh)\ \textbf{after}\ \ \textbf{dl}\ 5; s_8)$
$\mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, f_2 = !supply(sp, cr)\ \textbf{after}\ \ \textbf{dl}\ 15; s_{11})$
$= \mathcal{T}_{S_{sale}}(\{f_1 \mapsto \{wh\}\}, \{\{wh\} \mapsto 0 \cdot \langle c_{pack}, 0\rangle\}, rt, 0, 0, \textbf{wait}(f_1))$
$\mathcal{T}_{S_{sale}}(\{f_2 \mapsto \{rt, sp, cr\}\}, \emptyset, rt, c_{supply}, 0, \textbf{wait}(f_2))$
$= (\emptyset, \emptyset, 0, max(0, 0 \cdot \langle c_{pack}, 0\rangle))$
$(\emptyset, \emptyset, 0, c_{supply})$
$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}), f_3 = !deliver(cr)\ \textbf{after}\ \ \textbf{dl}\ 10; s_{13})$
$= \mathcal{T}_{S_{sale}}(\{f_3 \mapsto \{rt, sp, cr\}\}, \emptyset, rt, c_{deliver}, max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}), \textbf{wait}(f_3))$
$= (\emptyset, \emptyset, 0,\ max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}) + c_{deliver}\ )$

Translation of method *pack* :
$= \mathcal{T}_{S_{pack}}(\emptyset, \emptyset, wh, 0, 0, \textbf{cost}(k_1))$
$= (\emptyset, \emptyset, 0,\ k_1\ )$

Translation of method *supply* :
$= \mathcal{T}_{S_{supply}}(\emptyset, \emptyset, cr, 0, 0, \textbf{cost}(k_2); s_{23}; s_{24})$
$= \mathcal{T}_{S_{supply}}(\emptyset, \emptyset, cr, 0, k_2, f_4 = !deliver(cr)\ \textbf{after}\ \ \textbf{dl}\ 10; s_{24})$
$= \mathcal{T}_{S_{supply}}(\{f_4 \mapsto \{cr\}\}, \emptyset, cr, c_{deliver}, k_2, \textbf{wait}(f_4))$
$= (\emptyset, \emptyset, 0,\ k_2 + c_{deliver}\ )$

Translation of method *deliver* :
$= \mathcal{T}_{S_{pack}}(\emptyset, \emptyset, cr, 0, 0, r = \textbf{hold}(\{Driver, VanDriver, 5\}, \{Van, Delivery, 1500\}); s_{30})$
$= \mathcal{T}_{S_{pack}}(\emptyset, \emptyset, cr, 0, 0, \textbf{cost}(k_3); s_{31})$
$= \mathcal{T}_{S_{pack}}(\emptyset, \emptyset, cr, 0, k_3, \textbf{release}(r))$
$= (\emptyset, \emptyset, 0,\ k_3\ )$

Translation of method *main* :
$= \mathcal{T}_{S_{main}}(\emptyset, \emptyset, o, 0, 0, \texttt{Retailer}\ rt = \textbf{new}\ \texttt{Retailer};\ \texttt{Supplier}\ sp = \textbf{new}\ \texttt{Supplier};\ \texttt{Courier}\ cr = \textbf{new}\ \texttt{Courier}; s_{36})$
$= \mathcal{T}_{S_{main}}(\emptyset, \emptyset, o, 0, 0, sale(rt, sp, cr)\ \textbf{after}\ \ \textbf{dl}\ \texttt{null})$
$= (\emptyset, \emptyset, 0,\ c_{sale}\ )$

We notice that for each method the resulting translation environment Ψ is always empty, and t_a is always equal to 0 because every asynchronous method invocation is always synchronised within the caller method body.

5. Properties

The correctness of our analysis relies on the property that the execution time never rises throughout transitions. Therefore, the cost of the program in the initial configuration over-approximates the cost of each computation.

Cost Program. The cost of a program is calculated by solving a set of equations. Let a cost program be an equation system of the form:

$$\begin{array}{rcl} eq_{m_i} & = & exp_i \\ eq_{main} & = & exp_{main} \end{array}$$

where m_i are the method names and $1 \leq i \leq n$, exp_i and exp_{main} are cost expressions. The solution of the above cost program is the closed-form upper bound for the equation eq_{main}, which is a main method of the program.

Definition 5.1 *(Cost of Program).* Let $\mathcal{P}=(R\ \overline{C}\ \{\overline{T\ x;}\ s\})$ be an $\mathcal{R}$PL program, where

$$\begin{aligned}\overline{C} \ = \ & \textbf{class } C_1\{\overline{T\ x;}\ B\ m_1(\overline{T\ y})\{\overline{T'\ x;}\ s_1\}\ldots\} \\ & \vdots \\ & \textbf{class } C_j\{\overline{T\ x;}\ B\ m_k(\overline{T\ y})\{\overline{T'\ x;}\ s_1\}\ldots B\ m_n(\overline{T\ y})\{\overline{T'\ x;}\ s_n\}\}\end{aligned}$$

Then for every $1 \leq i \leq n$ and $1 \leq j \leq m$, let

1. $S_i = \texttt{sschem}(\{\{o_i, \overline{o'}\}\}, s_i, o_i)$
2. $eq_{m_i} = t_i$, where $\mathcal{T}_{S_i}(\emptyset, \emptyset, o_i, 0, 0, s_i) = (I_i, \Psi_i, t_a, t_i)$
3. $S_{main} = \texttt{sschem}(\{\{o_{main}\}\}, s, o_{main})$ and $\mathcal{T}_{S_{main}}(\emptyset, \emptyset, o_{main}, 0, 0, s) = (I, \Psi, t_a, t_{main})$

Let $eq(\mathcal{P})$ be the cost program $(eq_{m_1} = t_1,\ \ldots,\ eq_{m_n} = t_n,\ eq_{main} = t_{main})$. A cost solution of $\mathcal{P}$, named $\mathcal{U}(\mathcal{P})$, is the closed-form solution of the equation eq_{main} in $eq(\mathcal{P})$.

For all methods, we produce cost equations that associates the method's cost to the cost of its last statement, $eq_{m_i} = t_i$. Similarly, we produce one additional equation for the cost of the main method eq_{main} and its closed-form solution over-approximates the computational time of $\mathcal{R}$PL program.

Example 5.1. The cost program of Fig. 11 is shown as follows, where each cost expression is computed in Example 4.1.

$$\begin{aligned}&eq_{sale} = max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}) + c_{deliver}, \quad eq_{pack} = k_1, \\ &eq_{supply} = k_2 + c_{deliver}, \quad eq_{deliver} = k_3, \quad eq_{main} = c_{sale}.\end{aligned}$$

Correctness Property. The correctness of our analysis follows the theorem below.

Theorem 1 *(Correctness of Analysis). Let $\mathcal{P}$ be an $\mathcal{R}$PL program, whose initial configuration is cn, and $\mathcal{U}(\mathcal{P})$ be the closed-form solution of $\mathcal{P}$. If $cn \Rightarrow^* cn'$, then $\texttt{time}(cn') \leq \mathcal{U}(\mathcal{P})$.*

Proof. The proof of Theorem 1 is similar to the one proven in [31]. The main idea is to first extend function $\mathcal{T}$ for runtime configurations, and to define the cost of a computation $cn \Rightarrow^* cn'$, written as $\texttt{time}(cn \Rightarrow^* cn')$, to be the sum of the labels of the transitions, and to show that $\mathcal{U}(\mathcal{P})$ is a solution of $\mathcal{T}(cn)$, then $\mathcal{U}(\mathcal{P}) - \texttt{time}(cn \Rightarrow^* cn')$ is a solution of $\mathcal{T}(cn')$. $\square$

6. Conclusion

We have presented in this paper a formal language $\mathcal{R}$PL that can be used to model cross-organisational workflows consisting of concurrently running workflows. We used an example to show how the language can be employed to couple these concurrent workflows by means of resources and task dependencies. We also proposed a static analysis to over-approximate the computational time of an $\mathcal{R}$PL program. We also presented a proof sketch of the correctness of the proposed analysis.

As for the immediate next steps, we plan to implement a graphical user interface that allows planners to design workflows graphically that can be translated to $\mathcal{R}$PL models.

In addition, we plan to extend our analysis to under-approximate the cost of workflows. The idea of this extension can be roughly organised in a threefold modification. Firstly, for the case of a method invocation where the callee belongs to the same synchronisation set of the carrier, we add cost of only this invocation to the current cost instead of adding the cost of all the methods executing on the objects residing in the same synchronisation set. Secondly, in the case of a method call whose callee does not belong to the same synchronisation set of the carrier, we add the maximum cost between the cost of the invoked method and the cost accumulated from the point where the method is invoked until it is synchronised in the method currently being analysed. Furthermore, in the case of a conditional statement, the cost will be the minimum between the cost of the branches.

Furthermore, we intend to develop verification techniques to ensure the correctness of workflow models in $\mathcal{R}$PL for cross-organisational workflows. A reasonable starting point is to investigate how to extend KeY-ABS [15], a deductive verification tool for ABS, to support $\mathcal{R}$PL. The presented language is intended to be the first step towards the automation of cross-organisational workflow planning.

To achieve this long-term goal, we plan to implement a workflow modelling framework with the support of cost analysis. In this framework, planners can design workflows, update workflows, and simulate the execution of the workflows. By connecting the cost analysis to a constraint solver, the planner can estimate the overall execution time of collaborative workflows and see the effect of any changes in the resource allocation and task dependency. We foresee that such a framework can eventually contribute to automating planning for cross-organisational workflows.

CRediT authorship contribution statement

Muhammad Rizwan Ali: Conceptualization, Formal analysis, Methodology, Validation, Writing – original draft. **Yngve Lamo:** Supervision, Writing – review & editing. **Violet Ka I Pun:** Conceptualization, Formal analysis, Methodology, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] W.M. van der Aalst, The application of Petri nets to workflow management, J. Circuits Syst. Comput. 8 (1998) 21–66.
[2] W.M. van der Aalst, Loosely coupled interorganizational workflows:: modeling and analyzing workflows crossing organizational boundaries, Inf. Manag. 37 (2000) 67–75.
[3] W.M. van der Aalst, A.H. ter Hofstede, YAWL: Yet Another Workflow Language, Inf. Syst. 30 (2005) 245–275.
[4] G.A. Agha, Actors: a model of concurrent computation in distributed systems, Technical Report, Massachusetts Inst. of Tech. Cambridge Artificial Intelligence Lab., 1985.
[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-form upper bounds in static cost analysis, J. Automat. Reason. 46 (2011) 161–203.
[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, in: Quantitative Aspects of Programming Languages (QAPL 2010), Theor. Comput. Sci. 413 (2012) 142–159.
[7] E. Albert, J. Correas, E.B. Johnsen, G. Román-Díez, Parallel cost analysis of distributed systems, in: S. Blazy, T. Jensen (Eds.), Static Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 275–292.
[8] M.R. Ali, V.K.I Pun, Cost analysis for an actor-based workflow modelling language, in: S. Campos, M. Minea (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2021, pp. 104–121.
[9] M.R. Ali, V.K.I Pun, Towards a resource-aware formal modelling language for workflow planning, in: L. Bellatreche, G. Chernishev, A. Corral, S. Ouchani, J. Vain (Eds.), Advances in Model and Data Engineering in the Digitalization Era, Springer International Publishing, Cham, 2021, pp. 251–258.
[10] G. Behrmann, A. David, K.G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks, Uppaal 4.0, 2006.
[11] A. Bog, F. Puhlmann, A Tool for the Simulation of π-Calculus Systems, Open.BPM, 2006.
[12] K. Bouchbout, Z. Alimazighi, Inter-organizational business processes modelling framework, in: ADBIS (2), Citeseer, 2011, pp. 45–54.
[13] M. Chinosi, A. Trombetta, Bpmn: an introduction to the standard, Comput. Stand. Interfaces 34 (2012) 124–134.
[14] R.M. Dijkman, M. Dumas, C. Ouyang, Formal semantics and automated analysis of bpmn process models, 2007.
[15] C.C. Din, R. Bubel, R. Hähnle, KeY-ABS: a deductive verification tool for the concurrent modelling language ABS, in: A.P. Felty, A. Middeldorp (Eds.), Intl. Conf. on Automated Deduction, Springer, 2015, pp. 517–526.
[16] P. Dourish, Process descriptions as organisational accounting devices: the dual use of workflow technologies, in: Proceedings of the 2001 Intl. ACM SIGGROUP Conf. on Supporting Group Work, 2001, pp. 52–60.
[17] M. Dumas, A.H.M. ter Hofstede, UML Activity Diagrams as a Workflow Specification Language, in: UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Springer Berlin Heidelberg, 2001, p. 76.
[18] A. Ermedahl, A Modular Tool Architecture for Worst-Case Execution Time Analysis, Ph.D. thesis, Uppsala University, Division of Computer Systems, Computer Systems, 2003.
[19] A. Flores-Montoya, R. Hähnle, Resource analysis of complex programs with cost equations, in: Asian Symposium on Programming Languages and Systems, Springer, 2014, pp. 275–295.
[20] E. Giachino, E.B. Johnsen, C. Laneve, K.I Pun, Time complexity of concurrent programs, in: C. Braga, P.C. Ölveczky (Eds.), Formal Aspects of Component Software, Springer International Publishing, Cham, 2016, pp. 199–216.
[21] N. Gronau, R. Korf, C. Müller, Kmdl–capturing, analysing and improving knowledge intensive business processes, in: Modeling and Analyzing Knowledge Intensive Business Processes with KMDL-Comprehensive Insights into Theory and Practice, 2012, pp. 195–222.
[22] A. Gustavsson, A. Ermedahl, B. Lisper, P. Pettersson, Towards WCET analysis of multicore architectures using UPPAAL, in: B. Lisper (Ed.), 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2010, pp. 101–112, http://drops.dagstuhl.de/opus/volltexte/2010/2830, the printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.
[23] A.A. Haider, A. Nadeem, Formal modelling languages to specify real-time systems: a survey, Int. J. Future Comput. Commun. 2 (2013).
[24] M.B. Hassen, M. Turki, F. Gargouri, Sensitive business processes: characteristics, representation, and evaluation of modeling approaches, Int. J. Strateg. Inf. Technol. Appl. 9 (2018) 41–77.
[25] J. Hoffmann, Z. Shao, Automatic static cost analysis for parallel programs, in: European Symposium on Programming Languages and Systems, Springer, 2015, pp. 132–157.
[26] A.H. ter Hofstede, W.M. van der Aalst, M. Adams, N. Russell, Modern Business Process Automation: YAWL and Its Support Environment, Springer Science & Business Media, 2009.
[27] K. Jensen, A brief introduction to coloured Petri nets, in: E. Brinksma (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 203–208.
[28] K. Jensen, L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer Science & Business Media, 2009.
[29] E.B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: a core language for Abstract Behavioral Specification, in: Intl. Symposium on Formal Methods for Components and Objects, Springer, 2010, pp. 142–164.
[30] W. Jost, K. Wagner, The ARIS Toolset, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 15–31.
[31] C. Laneve, M. Lienhardt, K.I Pun, G. Román-Díez, Time analysis of actor programs, J. Log. Algebraic Methods Program. 105 (2019) 1–27.
[32] J. Meng, S.Y. Su, H. Lam, A. Helal, J. Xian, X. Liu, S. Yang, Dynaflow: a dynamic inter-organisational workflow management system, Int. J. Bus. Process Integr. Manag. 1 (2006) 101–115.
[33] F.F. Oliveira, J.C. Antunes, R.S. Guizzardi, Towards a collaboration ontology, in: Proc. of the Second Brazilian Workshop on Ontologies and Metamodels for Software and Data Engineering, João Pessoa, 2007.
[34] M.A. Ould, Business Processes: Modelling and Analysis for Re-engineering and Improvement, Wiley, 1995.
[35] C. Ouyang, M. Dumas, A.H. ter Hofstede, W.M. van der Aalst, From BPMN process models to BPEL web services, in: 2006 IEEE International Conference on Web Services (ICWS'06), IEEE, 2006, pp. 285–292.

[36] F. Puhlmann, M. Weske, Using the π-calculus for formalizing workflow patterns, in: International Conference on Business Process Management, Springer, 2005, pp. 153–168.
[37] D. Sundmark, Structural System-Level Testing of Embedded Real-Time Systems, Ph.D. thesis, Mälardalen University, Department of Computer Science and Electronics, 2008.
[38] K. Wagner, J. Klueckmann, Business Process Design as the Basis for Compliance Management, Enterprise Architecture and Business Rules, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 117–127.
[39] R. Woitsch, D. Karagiannis, Process oriented knowledge management: a service based approach, J. Univers. Comput. Sci. 11 (2005) 565–588, https://doi.org/10.3217/jucs-011-04-0565.
[40] L. Xu, H. Liu, S. Wang, K. Wang, Modelling and analysis techniques for cross-organizational workflow systems, Syst. Res. Behav. Sci. 26 (2009) 367–389.
[41] H. Yan, P. Chen, Research and application of workflow engine based on probabilistic timed automata of industrial internet of things, in: 2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP), 2021, pp. 1136–1139.