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 re-structuring 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

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-

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

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.

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.

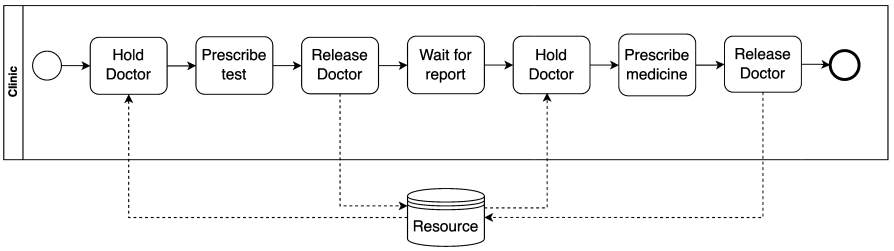

Fig. 3. A simple example.

```
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.

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

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

$$\frac{x \in dom(a) \quad v = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid x = e; s\},q) \rightarrow obj(o,a[x \mapsto v],\{l \mid s\},q)}\ (\textsc{Field-Assign}) \qquad \frac{x \in dom(l) \quad v = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid x = e; s\},q) \rightarrow obj(o,a,\{l[x \mapsto v] \mid s\},q)}\ (\textsc{Local-Assign})$$

$$\frac{true = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid \textbf{if } (e)\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\},q) \rightarrow obj(o,a,\{l \mid s_1; s\},q)}\ (\textsc{Cond-True}) \qquad \frac{false = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid \textbf{if } (e)\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\},q) \rightarrow obj(o,a,\{l \mid s_2; s\},q)}\ (\textsc{Cond-False})$$

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

$$\frac{o' = \texttt{fresh}() \quad a' = \texttt{atts}(C,o')}{obj(o,a,\{l \mid x = \textbf{new } C; s\},q) \rightarrow obj(o,a,\{l \mid x = o'; s\},q)\ obj(o',a',\texttt{idle},\emptyset)}\ (\textsc{New-Object}) \qquad \frac{v = [\![e]\!]_{(a \circ l)} \quad f = l(destiny)}{obj(o,a,\{l \mid \textbf{return } e; s\},q)\ fut(f,\bot) \rightarrow obj(o,a,\{l \mid s\},q)\ fut(f,v)}\ (\textsc{Return})$$

$$obj(o,a,\{l \mid \textbf{suspend}; s\},q) \rightarrow obj(o,a,\texttt{idle},q \cup \{l \mid s\})\ (\textsc{Suspend}) \qquad obj(o,a,\{l \mid \textbf{skip}; s\},q) \rightarrow obj(o,a,\{l \mid s\},q)\ (\textsc{Skip})$$

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

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 $[\![e']\!]_{(a \circ l)} > 0$.

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 $[\![e]\!]_{(a \circ l)}$ 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 `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 `idle`. If an object is `idle`, the rule Activate `selects` a suspended process to become active in an idle object, where the `select` function is defined as follows:

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

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.

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

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

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

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

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

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

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

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

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

$$\frac{rid = \{r\} \cup rid' \quad res(r) = (false,\mathcal{Q}) \quad res' = res[r \mapsto (true,\mathcal{Q})]}{obj(o,a,\{l \mid \textbf{release}(rid);s\},q)\ res \rightarrow obj(o,a,\{l \mid \textbf{release}(rid');s\},q)\ res'}\ (\textsc{Release-Resource-1})$$

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

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

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.

$$\frac{[\![e]\!]_{(a,o,l)} = 0}{obj(o,a,\{l \mid \textbf{cost}(e);s\},q) \rightarrow obj(o,a,\{l \mid s\},q)}\ (\textsc{Cost})$$

$$\frac{\texttt{strongstable}_t(cn) \quad cn' = \texttt{checkDl}(cn,t) \quad error \notin cn'}{cn \rightarrow \texttt{advance}(cn',t)}\ (\textsc{Tick-Ok})$$

$$\frac{\texttt{strongstable}_t(cn) \quad cn' = \texttt{checkDl}(cn,t) \quad error \in cn'}{cn \rightarrow error}\ (\textsc{Tick-Miss})$$

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

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

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 $\texttt{stable}_t(cn)$, 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 $[\![e]\!]_{(a,o,l)} \geq t$,
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. $p = \{l \mid x = m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\}$, or
      $p = \{l \mid x = !m(x',\overline{e})\ \textbf{after}\ \overline{fs}\ \textbf{dl}\ e';s\}$, where $\overline{fs} = false$.

A configuration cn is *strongly t-stable*, written as $\texttt{strongstable}_t(cn)$, if it is t-stable and there is an object $obj(o,a,\{l \mid \textbf{cost}(e);s\},q)$ with $[\![e]\!]_{(a,o,l)} = t$. 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*:

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

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:

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

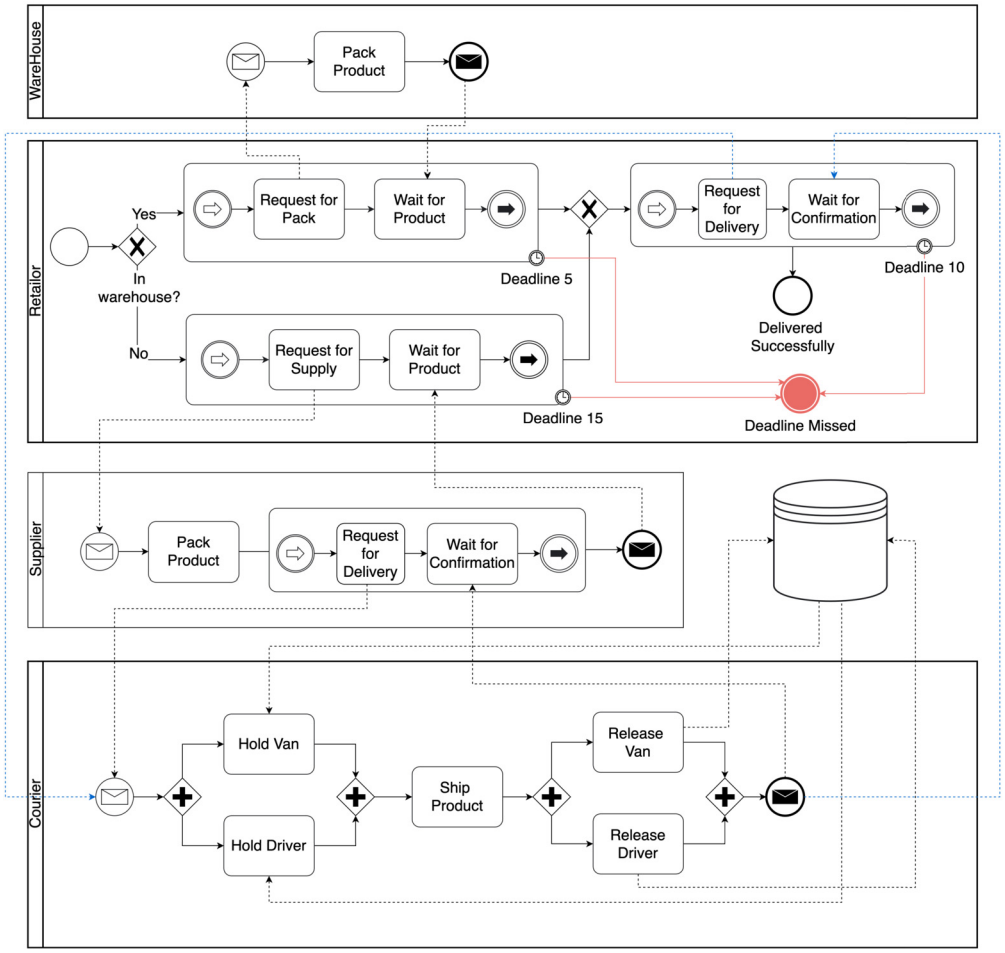

Fig. 10. An illustrative example.

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 $\texttt{time}(cn)$, 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 $S_m = \texttt{sschem}(\{\{o, \overline{o'}\}\}, s, o)$, 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.

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

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	$0 \cdot \langle c_{pack}, 0\rangle \;||\; 0 \cdot \langle c_{supply}, 0\rangle$
m_{sale}	12	$(0 \cdot \langle c_{pack}, 0\rangle \;||\; 0 \cdot \langle c_{supply}, 0\rangle) \cdot \langle c_{deliver}, 0\rangle$
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 $\mathbf{trans}_{S_m}(I, \Psi, x, t_a, t, \{f\})$.

$$
\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} \\
& \quad \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}\}, \\
& \quad \text{and,} \\
& \quad mt_a = max(t_a(\mathcal{D})), \text{ and} \\
& \quad 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} \\
& \quad \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}\}, \\
& \quad \text{and,} \\
& \quad mt_a = max(t_a(\mathcal{D})), \text{ and} \\
& \quad 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} \\
& \quad \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}\}, \\
& \quad \text{and,} \\
& \quad mt_a = max(t_a(\mathcal{D})), \text{ and} \\
& \quad mt = max(t(\mathcal{D})) \\
& \quad \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.

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

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* :

$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, if\,(In - Warehouse)\{s_7; s_8\}\ else\ \{s_{10}; s_{11}\}\ s_{12})$

$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, f_1 = !pack(wh)\ \textbf{after}\ \ \textbf{dl}\ 5; s_8)$
$\quad \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, 0, f_2 = !supply(sp, cr)\ \textbf{after}\ \ \textbf{dl}\ 15; s_{11})$

$= \mathcal{T}_{S_{sale}}(\{f_1 \mapsto \{wh\}\}, \{\{wh\} \mapsto 0 \cdot \langle c_{pack}, 0\rangle\}, rt, 0, 0, \textbf{wait}(f_1))$
$\quad \mathcal{T}_{S_{sale}}(\{f_2 \mapsto \{rt, sp, cr\}\}, \emptyset, rt, c_{supply}, 0, \textbf{wait}(f_2))$

$= (\emptyset, \emptyset, 0, max(0, 0 \cdot \langle c_{pack}, 0\rangle))$
$\quad (\emptyset, \emptyset, 0, c_{supply})$

$= \mathcal{T}_{S_{sale}}(\emptyset, \emptyset, rt, 0, max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}), f_3 = !deliver(cr)\ \textbf{after}\ \ \textbf{dl}\ 10; s_{13})$

$= \mathcal{T}_{S_{sale}}(\{f_3 \mapsto \{rt, sp, cr\}\}, \emptyset, rt, c_{deliver}, max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}), \textbf{wait}(f_3))$

$= (\emptyset, \emptyset, 0, \ max(max(0, 0 \cdot \langle c_{pack}, 0\rangle), c_{supply}) + c_{deliver}\ )$

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:

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

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 $1 \le i \le n$ and $1 \le j \le m$, 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 $\texttt{time}(cn') \le \mathcal{U}(\mathcal{P})$.*

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 $\mathcal{T}(cn')$. □

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.

Cost Analysis for an Actor-Based Workflow Modelling Language

Preprint · October 2021

CITATIONS
0

READS
24

2 authors:

Muhammad Rizwan Ali
Høgskulen på Vestlandet
11 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Violet Ka I Pun
Høgskulen på Vestlandet
63 PUBLICATIONS **313** CITATIONS
SEE PROFILE

Cost Analysis for an Actor-Based Workflow Modelling Language *

Muhammad Rizwan Ali[1] and Violet Ka I Pun[1]

Western Norway University of Applied Sciences, Norway
{mral,vpu}@hvl.no

Abstract. Workflow planning usually requires domain-specific knowledge from the planners, making it a relatively manual process. In addition, workflows are largely cross-organisational. As a result, minor modifications in the workflow of a collaborative partner may be propagated to other concurrently running workflows, which may result in significant adverse impacts. This paper presents a resource-sensitive formal modelling language, $\mathcal{R}$PL. The language has explicit notions for task dependencies, resource allocation and time advancement. The language allows the planners to estimate the effect of changes in collaborative workflows with respect to cost in terms of execution time. This paper proposes a static analysis for computing 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 cost equations.

Keywords: cross-organisational workflows · resource planning · formal modelling · static analysis

1 Introduction

Workflow management can be seen as an effective method of monitoring, managing, and improving business processes using IT assistance [1]. Workflow management systems (WMS) allow planners to create, manage, and execute workflows, as well as 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 [12]. 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 [26].

The adoption of cross-organisational workflow allows restructuring business processes beyond the limits of an organisation [2]. Cross-organisational workflows

* Partially supported by *CroFlow: Enabling Highly Automated Cross-Organisational Workflow Planning*, *Pathology services in the Western Norwegian Health Region – a center for applied digitization* and *SIRIUS – Centre for Scalable Data Access* (www.sirius-labs.no).

often comprise multiple concurrent workflows running in various departments within the same organisation or in different organisations. For example, the workflow of a retail company may involve a workflow of a supplier providing products and a workflow of a courier company delivering products to customers.

Furthermore, workflow planning often requires domain-specific knowledge to accomplish efficient resource allocation and task management, which makes planning cross-organisational workflow especially challenging. Additionally, modifying 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.

Workflow planning has been significantly digitalised and automated, and tools such as Process-Aware Information Systems (PAIS) [13] and Enterprise Resource Planning (ERP) systems have been developed to facilitate workflow planning. However, cross-organisational workflow planning remains a rather manual process as the current techniques and tools often lack domain-specific knowledge to support automation in workflow planning and updates. Moreover, the planners may only have limited domain knowledge and do not have a common understanding of all the collaborative workflows, which can be catastrophic, especially in the healthcare domain. Therefore, there is a need for an analysis that over-approximates the cost before any changes in the workflows are implemented. With the cost analysis, the planners can first simulates the changes in the design of workflows, including the task dependencies and resource allocation, and see the effect of the changes in terms of execution time before the changes are implemented in the workflow in practice.

In this paper, we first present a formal modelling language $\mathcal{R}$PL. The language has explicit notions for task dependencies, resource usage and time consumption, which allows the cross-organisational planners to couple various workflows through resources and task dependencies. A preliminary idea of the language is presented in [8]. In addition, we present a technique based on the work in [21] to statically over-approximate the worst execution time of the 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., [14,5]). This enables planners to estimate the effects of the workflows (and its possible changes) in terms of execution time before the actual implementation. 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 introduces the syntax and semantics of the language. Section 3 shows a static analysis to over-approximate the execution time of an $\mathcal{R}$PL program. Section 4 shows the correctness of analysis. Section 5 briefly discusses the related work. Finally, we summarise the paper and discuss possible future work in Section 6.

$$
\begin{array}{ll}
P ::= R\ \overline{Cl}\ \{\overline{T\ x};\ s\} & e ::= x \mid g \mid \textbf{this} \\
Cl ::= \textbf{class}\ C\ \{\overline{T\ x};\ \overline{M}\} & g ::= b \mid f? \mid g \wedge g \\
M ::= Sg\ \{\overline{T\ x};\ s\} & s ::= x = rhs \mid \textbf{skip} \mid \textbf{if}\ e\ \{s\} \mid \textbf{wait}(f) \mid \textbf{return}\ e \\
Sg ::= B\ m(\overline{T\ y}) & \quad \mid \textbf{hold}(\overline{r, e}) \mid \textbf{release}(\overline{r, e}) \mid \textbf{cost}(e) \mid s\ ;\ s \\
B ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit} & rhs ::= e \mid \textbf{new}\ C \mid f.\textbf{get} \\
T ::= C \mid B \mid \textbf{Fut}\langle B\rangle & \quad \mid m(x, \overline{e})\ \textbf{after}\ \overline{f?} \mid !m(x, \overline{e})\ \textbf{after}\ \overline{f?}
\end{array}
$$

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

2 Formal Workflow Modelling Language $\mathcal{R}$PL

In this section, we present a formal modelling language $\mathcal{R}$PL. The language is inspired by an active object language, ABS [19], and has a Java-like syntax and actor-based concurrency model. In an actor-based concurrency model [4], actors are primitives of concurrent computation. They can send a finite number of messages to other actors, spawn a finite number of new actors or modify their private state. A primary feature of the actor-based model is that one message is being processed per actor, preserving the invariants of an actor without locks.

$\mathcal{R}$PL uses explicit notions to express time advancement and to indicate resources required for each task (expressed as a method) and dependencies between tasks. Using cooperative scheduling of method activations, $\mathcal{R}$PL controls the internal interleaving of processes inside an object with explicit scheduling points.

2.1 The syntax of $\mathcal{R}$PL

The syntax of the language 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. Types T in $\mathcal{R}$PL are basic types B, including integer, boolean and unit type, a class C and future types $\textbf{Fut}\langle B\rangle$, which types asynchronous method invocations (see below).

Resources $R : r \mapsto v$ maps resource identifiers r to integer values v, indicating the number of resources r is available. A class declaration $\textbf{class}\ C\ \{\overline{T\ x};\ \overline{M}\}$ 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 $\overline{y}$. 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 $C\ o, \overline{C'\ o'}, \overline{T'\ x}$ where o is always the callee object identifier of the method of class C, $\overline{o'}$ are object identifiers of class $\overline{C'}$ and $\overline{x}$ are the remaining parameters. This assumption is the syntactic sugar that we use to realise the cost analysis introduced in Section 3. Expressions e include guards g, variables x and self-identifier **this**. A guard g allows a process to release control

of an object. It can be boolean conditions b, return tests $f?$ checking if the future variable f is resolved, or a conjunction of guards.

Statements include sequential composition, assignment, **if**, **skip**, and **return** are standard. Iterative loops are not included in the language, but can be implemented with recursion. $\mathcal{R}$PL uses **hold**$(\overline{r,e})$ and **release**$(\overline{r,e})$ to acquire and return e number of resources r. Statement **wait**(f) suspends the current process until future f is resolved, while other processes in the same object can be scheduled for execution. Statement **cost**(e), the only term in $\mathcal{R}$PL that consumes time, represents e units of time advancement.

The right-hand side *rhs* of an assignment includes expressions e, object creation **new** C, method invocations and synchronisation. Communication in $\mathcal{R}$PL is based on method calls, which can be either synchronous, written as $m(x,\overline{e})$ **after** $\overline{f?}$, or asynchronous, written as $!m(x,\overline{e})$ **after** $\overline{f?}$, where x is the callee object and $\overline{f?}$ is a sequence of futures that must be resolved prior to invoking method m. 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. An asynchronous method invocation is associated to a future variable of type **Fut**$\langle B\rangle$, where B is the return type of the invoked method. Moreover, the expression f.**get** blocks all execution in the object until future f is resolved.

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. The caller object will only be blocked if it tries to retrieve the value of the future with a **get** statement.

Fig. 2 shows a simple program in $\mathcal{R}$PL. The code snippet captures a simple collaboration between the workflows of a retail, a supplier and a courier company. Line 1 models the available resources. Lines 2–10 define a retail sale workflow. First, a request to the supplier for product supply is made asynchronously with associated future f_1 on Line 7. While waiting for the product (until f_1 is resolved), the retailer can continue with other tasks. After getting the product from the supplier (f_1 is resolved), it is sent to the customer by utilising the services of a courier company (Line 8). Lines 11–16 define the deliver workflow of the courier company. A driver and a vehicle (resources) are first acquired to deliver the product (Line 13). Line 14 depicts the time taken for delivery. Afterwards, the acquired resources are released (Line 15). For simplicity, we do not show the implementation of the supply workflow.

```
1  [Driver ↦ 5, Vehicle ↦ 3]
2  class Retail {
3    Unit sale(Retail o, Int ord) {
4      Fut<Bool> f1;
5      Supplier sp = new Supplier;
6      Courier cr = new Courier;
7      f1 = !supply(sp,ord) after;
8      Unit x = deliver(cr,ord,10) after f1?;
9    }
10 }
11 class Courier {
12   Unit deliver(Courier o, Int ord, Int t) {
13     hold(Driver,1)(Vehicle,1);
14     cost(t);
15     release(Driver,1)(Vehicle,1);
16 }}
```

Fig. 2. A simple example.

2.2 The Semantics of $\mathcal{R}$PL

To understand how time advances in $\mathcal{R}$PL and the cost analysis later, 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 with the runtime syntax defined in Fig. 3.

$$
\begin{array}{ll}
cn ::= \varepsilon \mid res \mid obj(o,a,p,q) \mid fut(f,val) & act ::= \varepsilon \mid o \\
\quad \mid invoc(o,f,m,\overline{v}) \mid cn\ cn & val ::= v \mid \bot \\
p ::= \texttt{idle} \mid \{l \mid s\} & res ::= [\ r \mapsto v\ ] \\
q ::= \emptyset \mid \{l \mid s\} \mid q\ \ q & a ::= [\ldots, x \mapsto v, \ldots] \\
s ::= \textsf{cont}(f) \mid \ldots & v ::= o \mid f \mid b \mid k
\end{array}
$$

Fig. 3. Runtime syntax of $\mathcal{R}$PL

A configuration cn includes futures, objects, message invocations, and resources. An empty configuration is ε, and whitespace denotes the associative and commutative union operator on configurations. A future $fut(f, val)$ holds a future identifier f and a return value val, where $\bot$ indicates that future has not been resolved.

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, written as $\{l \mid s\}$, has local variable bindings l and a statement s. A message invocation is a term $invoc(o, f, m, \overline{v})$, where o is a callee object, m a method name, f a future to which method m returns, and $\overline{v}$ the set of actual parameter values for m. Resources res is a mapping from resource identifier r to the number of resources. The statement $\textsf{cont}(f)$ controls the scheduling when a synchronous call completes its execution, returning control to the caller. Values v include object, future identifier, and Boolean, Integer or constant values.

We discuss a selection of the semantics rules of $\mathcal{R}$PL (see Figs. 4 and 5) that are relevant to the analysis later. The rest of the semantics is standard, and can be found in the accompanying technical report [7]. In the semantics, we use the auxiliary functions $dom(l)$ and $dom(a)$ to return the domain of l and a, respectively. The evaluation function $[\![e]\!]_{(aol)}$ returns the value of e by computing the expressions and retrieving the value of identifiers stored either in a or l. Moreover, the function $\texttt{atts}(C, o)$ is used to create an object of a class C, which binds `this` to o, and the function $\texttt{bind}(o, f, m, \overline{v}, C)$ 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:

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

The semantics in Figs. 4 and 5 includes object creation, communication, task dependencies, resource management and time advancement. For clarity, we use $\mathbb{F}$ to represent all the futures in the configuration in the semantics.

$$\frac{o' = \texttt{fresh}() \quad a' = \texttt{atts}(C, o')}{obj(o, a, \{l \mid x = \textbf{new}\ C; s\}, q) \rightarrow obj(o, a, \{l \mid x = o'; s\}, q)\ obj(o', a', \texttt{idle}, \emptyset)}\ (\textsc{New-Object})$$

$$\frac{\forall\ f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad o' = [\![e]\!]_{(a \circ l)} \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l \mid x = f'; s\}, q)\ invoc(o', f', m, \overline{v})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Async-Call})$$

$$\frac{v \neq \bot}{obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid x = v; s\}, q)\ fut(f, v)}\ (\textsc{Get})$$

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

$$\frac{v \neq \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid s\}, q)\ fut(f, v)}\ (\textsc{Wait-True})$$

$$\frac{v = \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid \textbf{wait}(f); s\})\ fut(f, v)}\ (\textsc{Wait-False})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o' = [\![e]\!]_{(a \circ l)} \quad o \neq o' \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F} \rightarrow obj(o, a, \{l \mid f' = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; x = f'.\textbf{get}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F}}\ (\textsc{Sync-Call})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad f'' = l(destiny) \quad f' = \texttt{fresh}() \quad \{l' \mid s'\} = \texttt{bind}(o, f', m, \overline{v}, \texttt{class}(o))}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l' \mid s'; \textbf{cont}(f'')\}, q \cup \{l \mid x = f'.\textbf{get}; s\})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Self-Sync-Call})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Async-Call})$$

$$\frac{f'' = l(destiny)}{obj(o, a, \{l' \mid \textbf{cont}(f''), q \cup \{l|s\}) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Sync-Return-Sched})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Sync-Call})$$

$$\frac{[\![e]\!]_{(a \circ l)} = 0}{obj(o, a, \{l \mid \textbf{cost}(e); s\}, q) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Cost})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v \geq 0 \quad where\ v = res(r) - [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{hold}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Hold})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v = res(r) + [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{release}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Release})$$

Fig. 4. A selection of semantics – Part 1

$$\frac{\texttt{strongstable}_{\texttt{t}}(cn)}{cn \to \Phi(cn,t)}\ (\textsc{Tick})$$

where, $\Phi(cn,t) =$

$$\begin{cases} obj(o,a,\{\ l' \mid \textbf{cost}(k);s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{cost}(e);s\},q)\ cn' \\ & \text{and } k = [\![e]\!]_{(a \circ l)} - t \\ obj(o,a,\{l \mid \textbf{hold}(\overline{r,e});s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{hold}(\overline{r,e});s\},q)\ cn' \\ obj(o,a,\{l \mid x = e.\textbf{get};s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid x = e.\textbf{get};s\},q)\ cn' \\ obj(o,a,\texttt{idle},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\texttt{idle},q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

Fig. 5. A selection of semantics – Part 2

Rule Wait-False suspends the active process, leaving the object idle if f is not resolved, otherwise Wait-True consumes $\textbf{wait}(f)$. Rule New-Object creates a new object. Rule Get retrieves the value of future f if it is resolved; the reduction on this object is blocked otherwise.

Rules Async-Call and Sync-Call handle the communication between objects through method invocations. To ensure the task dependencies between method calls, the rules first check if all the futures on which the method call depends exists, i.e., if $\overline{f}$ can be found in $\mathbb{F}$ and check if they are resolved. Rule Async-Call creates an invocation message to o' with a fresh unresolved future f', method name m, and actual parameters $\overline{v}$. Rule Self-Sync-Call directly transfers 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-Call specifies a synchronous call to another object, which is replaced by an asynchronous call followed by a **get** statement. In case one of the futures that a synchronous (or asynchronous) method invocations depends on is not yet resolved, the process will be suspended (see Rules (Wait-Async-Call) and (Wait-Sync-Call)). Rules Hold and Release control the resource acquisition and return. Note that it is required to have all the acquired resources to be available in order to consume the **hold** statement; otherwise, the process will be blocked.

In $\mathcal{R}$pl, the unique statement that consumes time is $\textbf{cost}(e)$. Rule Cost specifies a trivial case when e evaluates to 0. When the configuration cn reaches a *stable state*, no other transition is possible except those evaluating the $\textbf{cost}(e)$ statement where e evaluates to some $t \leq 0$, then time advances by the smallest value required to let at least one process execute. To formalize this semantics, we first define stability in Definition 1.

Definition 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. $obj(o,a,\{l \mid x = e.\textbf{get};s\},q)$ *where* $[\![e]\!]_{(a \circ l)} = f$ *and* $fut(f,\bot) \in cn$,
2. $obj(o,a,\{l \mid \textbf{cost}(e);s\},q)$ *where* $[\![e]\!]_{(a \circ l)} \geq t$,

3. $obj(o, a, \{l \mid \textbf{hold}(\overline{r, e}); s\}, q)$ *with* $res \in cn$,
   where $\exists (r, e) \in \overline{(r, e)}$ *s.t.* $r \in dom(res)$ *and* $res(r) - [\![e]\!]_{(aol)} \leq 0$,
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. $p = \{l \mid x = m(e, \overline{e'})$ **after** $\overline{f?}; s\}$, *or* $p = \{l \mid x = !m(e, \overline{e'})$ **after** $\overline{f?}; s\}$,
      where $\exists f \in \overline{f}$ *s.t.* $fut(f, \bot) \in cn$.

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 $[\![e]\!]_{(aol)} = t$. Note that both t-stable and strongly t-stable configurations cannot proceed anymore because every object is stuck either on a $\textbf{cost}(e)$, on unresolved futures, or waiting for some resources. Rule Tick in Fig. 5 handles time advancement when cn is strongly t-stable by advancing time in cn for t units using $\Phi(cn, t)$.

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

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

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 $\stackrel{t}{\Rightarrow}$ is $\rightarrow^* \stackrel{t}{\rightarrow} \rightarrow^*$. A computation is $cn \stackrel{t_1}{\Longrightarrow} \ldots \stackrel{t_n}{\Longrightarrow} cn'$; that is, cn' is a configuration reachable from cn with either transitions $\rightarrow$ or $\stackrel{t}{\Rightarrow}$. When the time labels of transitions are not necessary, we also write $cn \Rightarrow^* cn'$.

Definition 2. *The computational time of* $cn \stackrel{t_1}{\Longrightarrow} \ldots \stackrel{t_n}{\Longrightarrow} cn'$ *is* $t_1 + \cdots + t_n$.

The computational time of a configuration cn, written as $\texttt{time}(cn)$, 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 Analysis of $\mathcal{R}$PL program

In this section, we describe the cost analysis for an $\mathcal{R}$PL program, which 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. 6 to illustrate the idea of the analysis. Our analysis assumes all $\mathcal{R}$PL programs terminate and all invoked methods are synchronised. It extends the analysis presented in [21] and to handle a more expressive language with explicit notion of task dependencies and resource allocations.

A cost equation results in a cost expression *exp* that has the following syntax:

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

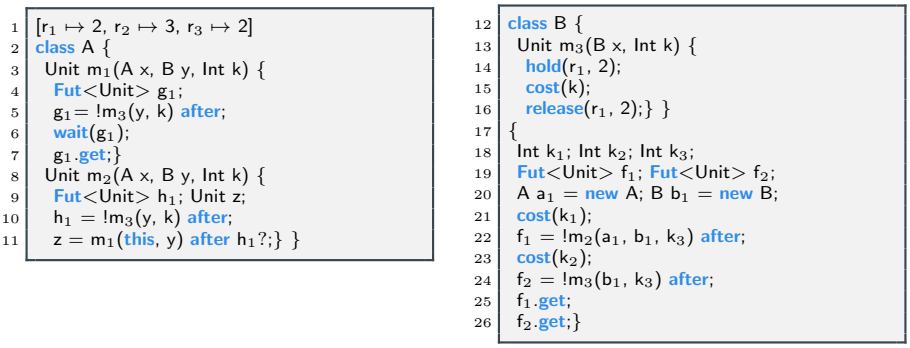

Fig. 6. A running example of an $\mathcal{R}$PL program.

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 for the finishing time of all of its processes, which gives rise to an upper bound to 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.

3.1 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 [21], 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 $S_m = \texttt{sschem}(\{\{o, \overline{o'}\}\}, s, o)$, which is defined in Definition 3.

Definition 3 (Synchronisation Schema Function). *Let S be a synchronisation schema, s a statement and o a carrier object which is executing s.*

$$\texttt{sschem}(S,s,o) = \begin{cases} S \oplus \{o', \overline{o''}\} & \text{if } s \text{ is } x = m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{f?} \\ & \quad \textit{or, } x = !m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{f'?} \\ \texttt{sschem}(S, s_1, o) & \text{if } s \text{ is } \textbf{if } e \ \{s_1\} \\ \texttt{sschem}(\texttt{sschem}(S, s', o), s'', o) & \text{if } s \text{ is } s'; s'' \\ S & \text{otherwise.} \end{cases}$$

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 methods defined in Fig. 6 are $S_{m_1} = \{\{x, y\}\}$, $S_{m_2} = \{\{x, y\}\}$, $S_{m_3} = \{\{x\}\}$, $S_{main} = \{\{o_{main}\}, \{a_1, b_1\}\}$.

3.2 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* [21], 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., $o' \notin S(o)$. 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** $\overline{f?}$, 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 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 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 $\mathcal{E}$ the time 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:

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

$$
\boldsymbol{\mathcal{T}}_{S_m}(I,\Psi,o,t_a,t,s) \;=\; \begin{cases}
1.\ \boldsymbol{\mathcal{T}}_{S_m}(I',\Psi',o,t'_a,t',s'') & \text{if } s \text{ is } s';s'', \text{and} \\
 & \quad (I',\Psi',t'_a,t') = \boldsymbol{\mathcal{T}}_{S_m}(I,\Psi,o,t_a,t,s') \\
2.\ (I,\Psi+e,t_a,t+e) & \text{if } s \text{ is } \textbf{cost}(e) \\
3.\ (I',\Psi',t'_a,t'+c_{m'}) & \text{if } s \text{ is } o = m'(o',\overline{e}) \textbf{ after } \overline{f?}, \text{ and} \\
 & \quad (I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\overline{f}) \\
4.\ (I'[f \mapsto S_m(o)],\Psi',t'_a+c_{m'},t') & \text{if } s \text{ is } f = !m'(o',\overline{e}) \textbf{ after } \overline{f'?},\ o' \in S_m(o), \text{and} \\
 & \quad (I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\overline{f'}) \\
5.\ (I'[f \mapsto S_m(o')],\Psi'[S_m(o') \mapsto \mathcal{E}\cdot\langle c_{m'},0\rangle],t'_a,t') & \\
 & \text{if } s \text{ is } f = m'(o',\overline{e}) \textbf{ after } \overline{f'?},\ o' \notin S_m(o), \text{and} \\
 & \quad (I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\overline{f'}), \text{where} \\
 & \quad \mathcal{E} = \begin{cases} \Psi'(S_m(o')) & \text{if } S_m(o') \in dom(\Psi') \\ t' & \text{otherwise.} \end{cases} \\
6.\ (I',\Psi',t'_a,t') & \text{if } s \text{ is } f.\textbf{get} \text{ or } \textbf{wait}(f), \text{ and} \\
 & \quad (I',\Psi',t'_a,t') = \textbf{trans}_{S_m}(I,\Psi,o,t_a,t,\{f\}) \\
7.\ (I',\Psi',max(t_a,t_{a_1}),max(t,t_1)) & \text{if } s \text{ is } \textbf{if } e\ \{s_1\}, \text{and} \\
 & \quad (I_1,\Psi_1,t_{a_1},t_1) = \boldsymbol{\mathcal{T}}_{S_m}(I,\Psi,o,t_a,t,s_1) \\
 & \quad I' = I \cup I_1, \text{ and} \\
 & \quad \Psi' = \texttt{upd}(\Psi,\Psi_1,I',dom(I')) \\
8.\ (I,\Psi,t_a,t) & \text{otherwise.}
\end{cases}
$$

Fig. 7. The translation function

The table below shows the accumulated costs of some of the statements declared in Fig. 6. The accumulated cost of Line 24 evaluates to $k_1 + max(c_{m_2}, k_2) + c_{m_3}$, which is the cost expression of the *main* method (c_{main}).

Method	*Line*	*Accumulated Cost*	*Method*	*Line*	*Accumulated Cost*
m_1	5	$0 \cdot \langle c_{m_3}, 0\rangle$	*main*	22	$k_1 \cdot \langle c_{m_2}, 0\rangle$
m_2	10	$0 \cdot \langle c_{m_3}, 0\rangle$	*main*	23	$k_1 \cdot \langle c_{m_2}, k_2\rangle$
m_3	15	k	*main*	24	$k_1 \cdot \langle c_{m_2}, k_2\rangle \cdot \langle c_{m_3}, 0\rangle$

3.3 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 3.1, 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.

Definition 4 (Translation Environment). *Translation environments, ranged over $\Psi, \Psi', \ldots$, is a mapping from synchronisation sets to their corresponding accumulated costs ($S_m(o) \mapsto \mathcal{E}$).*

Given a synchronisation schema of a method m, S_m, the translation function $\boldsymbol{\mathcal{T}}_{S_m}(I,\Psi,o,t_a,t,s)$ defined in Fig. 7 takes six parameters: I is a map from future

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

Fig. 8. The auxiliary translation function

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 and 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'. Each case of the function is explained below.

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: If s is a synchronous method invocation $m'(o', \overline{e})$ **after** $\overline{f?}$, since the method can only be invoked after the futures $\overline{f}$[1] have been resolved, we need to first compute the cost of all methods associating to $\overline{f?}$ with the auxiliary function $\textbf{trans}_{S_m}(I, \Psi, o, t_a, t, \overline{f})$ in Fig. 8 (see below for explanation). After computing the cost of executing the methods associating to $\overline{f}$, the cost of method m', $c_{m'}$, is added to the accumulated cost t'.

Case 4 & 5: The next two cases corresponds to s as an asynchronous method invocation $!m'(o', \overline{e})$ **after** $\overline{f?}$. Similar to **Case 3**, we first compute the cost of all methods associating to $\overline{f?}$. **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 t'_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 $\textbf{trans}_{S_m}(I, \Psi, x, t_a, t, \{f\})$.

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_1}, and the current cost t with the maximum of t and t_1.

[1] We refer $\overline{f}$ to a (possibly empty) set of futures by overloading the overline notation.

$$\texttt{upd}(\Psi_1, \Psi_2, I, F) =$$
$$\begin{cases} \Psi_1 & \text{if } F = \emptyset \ \vee \ \Psi_2 = \emptyset \\ \Psi_2 & \text{if } \Psi_1 = \emptyset \\ \texttt{upd}(\Psi_1[I(f) \mapsto max(\Psi_1(I(f)), \Psi_2(I(f)))], \Psi_2, I, F') & \\ & \text{if } F = F' \cup f \wedge I(f) \in dom(\Psi_1) \wedge I(f) \in dom(\Psi_2) \\ \texttt{upd}(\Psi_1, \Psi_2, I, F') & \text{if } F = F' \cup f \wedge I(f) \in dom(\Psi_1) \wedge I(f) \notin dom(\Psi_2) \\ \texttt{upd}(\Psi_1[I(f) \mapsto \Psi_2(I(f))], \Psi_2, I, F') & \text{if } F = F' \cup f \wedge I(f) \notin dom(\Psi_1) \wedge I(f) \in dom(\Psi_2) \end{cases}$$

Fig. 9. The auxiliary update function

The resulting I' is the union of I and I_1. We further update the translation environment with the auxiliary update function defined in Fig. 9.

The trans function. The auxilary function **trans** in Fig. 8 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 1. We show how the translation function can be applied on the methods defined in Fig. 6. Let $S = \{\{o\}, \{a_1, b_1\}\}$, $S_1 = \{\{x, y\}\}$, $S_2 = \{\{x, y\}\}$ and $S_3 = \{\{x\}\}$ (as computed in Section 3.2). We use s_i to indicate the sequence of statements of a method body starting from line i.

Translation of method m_1 : $\mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, 0, g_1 = !m_3(y,k)$ **after**; $s_6)$
$= \mathcal{T}_{S_1}(\{g_1 \mapsto \{x, y\}\}, \emptyset, x, c_{m_3}, 0,$ **wait**$(g_1); s_7)$
$= \mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, c_{m_3}, g_1.$**get**$)$
$= (\emptyset, \emptyset, 0, \boxed{c_{m_3}})$

Translation of method $m_2 : \mathcal{T}_{S_2}(\emptyset, \emptyset, x, 0, 0, h_1 = !m_3(y,k)$ **after**; $s_{11})$
$= \mathcal{T}_{S_2}(\{h_1 \mapsto \{x, y\}\}, \emptyset, x, c_{m_3}, 0, z = m_1($**this**$, y)$ **after** $h1?)$
$= (\emptyset, \emptyset, 0, \boxed{c_{m_3} + c_{m_1}})$

Translation of method $m_3 : \mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, 0,$ **hold**$(r_1, 2); s_{15})$
$= \mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, 0,$ **cost**$(k); s_{16})$
$= \mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, k,$ **release**$(r_1, 2))$
$= (\emptyset, \emptyset, 0, \boxed{k})$

Translation of method $main$:
$\mathcal{T}_S(\emptyset, \emptyset, o, 0, 0,$ `A` $a_1 =$ **new** `A`; `B` $b_1 =$ **new** `B`; $s_{21})$
$= \mathcal{T}_S(\emptyset, \emptyset, o, 0, 0,$ **cost**$(k_1); s_{22})$
$= \mathcal{T}_S(\emptyset, \emptyset, o, 0, k_1, f_1 = !m_2(a_1, b_1, k_3)$ **after**; $s_{23})$
$= \mathcal{T}_S(\{f_1 \mapsto \{a_1, b_1\}\}, \{\{a_1, b_1\} \mapsto k_1 \cdot \langle c_{m_2}, 0\rangle\}, o, 0, k_1,$ **cost**$(k_2); s_{24})$
$= \mathcal{T}_S(\{f_1 \mapsto \{a_1, b_1\}\}, \{\{a_1, b_1\} \mapsto k_1 \cdot \langle c_{m_2}, k_2\rangle\}, o, 0,$
$k_1 + k_2, f_2 = !m_3(b_1, k_3)$ **after**; $s_{25})$
$= \mathcal{T}_S(\{f_1 \mapsto \{a_1, b_1\}, f_2 \mapsto \{a_1, b_1\}\}, \{\{a_1, b_1\} \mapsto k_1 \cdot \langle c_{m_2}, k_2\rangle \cdot \langle c_{m_3}, 0\rangle\}, o, 0,$
$k_1 + k_2, f_1.$**get**; $s_{26})$
$= \mathcal{T}_S(\emptyset, \emptyset, o, 0, max(k_1 + k_2, k_1 \cdot \langle c_{m_2}, k_2\rangle \cdot \langle c_{m_3}, 0\rangle), f_2.$**get**$)$
$= (\emptyset, \emptyset, 0, \boxed{max(k_1 + k_2, k_1 \cdot \langle c_{m_2}, k_2\rangle \cdot \langle c_{m_3}, 0\rangle)})$

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.

4 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:

$$eq_{m_i} = exp_i$$
$$eq_{main} = exp_{main}$$

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 (Cost of Program). *Let* $\mathcal{P} = (R\ \overline{C}\ \{\overline{T\ x};\ s\})$ *be an* $\mathcal{R}$PL *program, where* $\overline{C} =$ **class** $C_1\ \{\overline{T\ x};\ B\ m_1(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\}$
$\vdots$
class $C_j\ \{\overline{T\ x};\ B\ m_k(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\ B\ m_n(\overline{T\ y})\{\overline{T'\ x};\ s_n\}\}$
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* $\boldsymbol{\mathcal{T}}_{S_i}(\emptyset, \emptyset, o_i, 0, 0, s_i) = (I_i, \Psi_i, t_a, t_i)$
3. $S_{main} = \texttt{sschem}(\{\{o_{main}\}\}, s, o_{main})$ *and*
   $\boldsymbol{\mathcal{T}}_{S_{main}}(\emptyset, \emptyset, o_{main}, 0, 0, s) = (I, \Psi, t_a, t_{main})$

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 2. The cost program of Fig. 6 is shown as follows, where each cost expression is computed in Example 1.

$$eq_{m_1} = c_{m_3}\ ,\quad eq_{m_2} = c_{m_3} + c_{m_1}\ ,\quad eq_{m_3} = k\ ,$$
$$eq_{main} = max(k_1 + k_2, k_1 \cdot \langle c_{m_2}, k_2 \rangle \cdot \langle c_{m_3}, 0 \rangle)\ .$$

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 (Sketch). The proof is similar to the one proven in [21]. The main idea is to first extend function $\boldsymbol{\mathcal{T}}$ 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 $\boldsymbol{\mathcal{T}}(cn)$, then $\mathcal{U}(\mathcal{P}) - \texttt{time}(cn \Rightarrow^* cn')$ is a solution of $\boldsymbol{\mathcal{T}}(cn')$.

5 Related Work

Comprehensive research has been performed on modelling business process workflows: BPEL [22] is an executable language for simulating process behaviour, whereas BPMN [24] uses a graphical notation to represent business process descriptions. Petri-nets [1] has been used to formalize both BPEL and BPMN [9,17]. Different formal approches based on e.g., pi-calculus [3], timed automata [16], CSP [25] have been developed to analyse and reason about models of business process workflows. Compared to our proposed approach, the main focus of these techniques is on intra-organisational workflows and have limited support for coordinating tasks and resources in workflows that are across organisational.

Approaches have been proposed to merge business process models, e.g., [15] presents an approach to merge two business processes based on Event-driven Process Chains [23], which has been implemented in the process mining framework ProM [11], and [20] describes a technique that generates a configurable business process with a pair of business processes as input. To the best of our knowledge, these techniques do not consider connecting workflows across organisations.

Numerous techniques have been introduced for static cost analysis. For example, [6] presents the first approach to the automatic cost analysis of object-oriented bytecode programs, [18] proposes the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. In [21], authors define a concurrent actor language with time. Also, they define a translation function that uses synchronisation sets to compute a cost equation function for each method definition. Compared to this techniques, this paper handles a more expressive language that is sensitive to task dependencies and resource consumption.

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 use 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 enrich the language such that the resource features, e.g., the experience and specialities, can be explicitly specified, and to extend the analysis to handle non-terminating programs. We also plan to develop an approach to associate workflow resources to ontology models. 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 [10], 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 and update workflows modelled in $\mathcal{R}$PL, 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 framework can eventually contribute to automating planning for cross-organisational workflows.

References

1. van der Aalst, W.M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers **8**(01), 21–66 (1998)
2. van der Aalst, W.M.: Loosely coupled interorganizational workflows:: modeling and analyzing workflows crossing organizational boundaries. Information & management **37**(2), 67–75 (2000)
3. Abouzaid, F.: A mapping from pi-calculus into BPEL. Frontiers in artificial intelligence and applications **143**, 235 (2006)

4. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1985)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. Journal of automated reasoning **46**(2), 161–203 (2011)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012), Quantitative Aspects of Programming Languages (QAPL 2010)
7. Ali, M.R., Pun, V.K.I: Cost Analysis for an Actor-Based Workflow Modelling Language (long version). Research Report 15, Western Norway Univ. of Applied Sciences (2021)
8. Ali, M.R., Pun, V.K.I: Towards a resource-aware formal modelling language for workflow planning. In: Intl. Conf. on Model and Data Engineering. Springer (To appear) (2021)
9. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri nets. Queensland Univ. of Technology, Tech. Rep. pp. 1–30 (2007)
10. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Intl. Conf. on Automated Deduction. LNCS, vol. 9195, pp. 517–526. Springer (2015)
11. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H., Weijters, A., van der Aalst, W.M.: The ProM framework: A new era in process mining tool support. In: Intl. Conf. on Application and Theory of Petri Nets. pp. 444–454. Springer (2005)
12. Dourish, P.: 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. pp. 52–60 (2001)
13. Dumas, M., van der Aalst, W.M., Ter Hofstede, A.H.: Process-aware information systems: bridging people and software through process technology. John Wiley & Sons (2005)
14. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Asian Symposium on Programming Languages and Systems. pp. 275–295. Springer (2014)
15. Gottschalk, F., van der Aalst, W.M., Jansen-Vullers, M.H.: Merging event-driven process chains. In: OTM Confederated Intl. Confs. On the Move to Meaningful Internet Systems. pp. 418–426. Springer (2008)
16. Gruhn, V., Laue, R.: Using timed model checking for verifying workflows. Computer Supported Activity Coordination **2005**, 75–88 (2005)
17. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: Intl. Conf. on Business Process Management. pp. 220–235. Springer (2005)
18. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: European Symposium on Programming Languages and Systems. pp. 132–157. Springer (2015)
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for Abstract Behavioral Specification. In: Intl. Symposium on Formal Methods for Components and Objects. pp. 142–164. Springer (2010)
20. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.: Merging business process models. In: OTM Confederated Intl. Confs." On the Move to Meaningful Internet Systems". pp. 96–113. Springer (2010)
21. Laneve, C., Lienhardt, M., Pun, K.I, Román-Díez, G.: Time analysis of actor programs. Journal of Logical and Algebraic Methods in Programming **105**, 1–27 (2019)
22. Matjaz Juric, Benny Mathew, P.S.: Business Process Execution Language for Web Services BPEL and BPEL4WS. Packt Publishing (2006)

23. Mendling, J.: Event-driven process chains (epc). In: Metrics for process models, pp. 17–57. Springer (2008)
24. OMG, B.P.M.: Notation (BPMN) Version 2.0 (2011)
25. Wong, P.Y., Gibbons, J.: Property specifications for workflow modelling. Science of Computer Programming **76**(10), 942–967 (2011)
26. Xu, L., Liu, H., Wang, S., Wang, K.: Modelling and analysis techniques for cross-organizational workflow systems. Systems Research and Behavioral Science: The Official Journal of the Intl. Federation for Systems Research **26**(3), 367–389 (2009)

Cost Analysis for an Actor-Based Workflow Modelling Language (Technical Report)

Technical Report · October 2021

CITATIONS
0

READS
14

2 authors:

Muhammad Rizwan Ali
Høgskulen på Vestlandet
11 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Violet Ka I Pun
Høgskulen på Vestlandet
63 PUBLICATIONS **313** CITATIONS
SEE PROFILE

Cost Analysis for an Actor-Based Workflow Modelling Language (Technical Report)

Muhammad Rizwan Ali[1] and Violet Ka I Pun[1]

Western Norway University of Applied Sciences, Norway
{mral,vpu}@hvl.no

Abstract. Workflow planning usually requires domain-specific knowledge from the planners, making it a relatively manual process. In addition, workflows are largely cross-organisational. As a result, minor modifications in the workflow of a collaborative partner may be propagated to other concurrently running workflows, which may result in significant adverse impacts. This paper presents a resource-sensitive formal modelling language, $\mathcal{R}$PL. The language has explicit notions for task dependencies, resource allocation and time advancement. The language allows the planners to estimate the effect of changes in collaborative workflows with respect to cost in terms of execution time. This paper proposes a static analysis for computing 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 cost equations.

Keywords: cross-organisational workflows · resource planning · formal modelling · static analysis

1 Introduction

Workflow management can be seen as an effective method of monitoring, managing, and improving business processes using IT assistance [1]. Workflow management systems (WMS) allow planners to create, manage, and execute workflows, as well as 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 [13]. 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 [27].

The adoption of cross-organisational workflow allows restructuring 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 in different organisations. For example, the workflow of a retail company may involve a workflow of a supplier providing products and a workflow of a courier company delivering products to customers.

Furthermore, workflow planning often requires domain-specific knowledge to accomplish efficient resource allocation and task management, which makes planning cross-organisational workflow especially challenging. Additionally, modifying 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.

Workflow planning has been significantly digitalised and automated, and tools such as Process-Aware Information Systems (PAIS) [14] and Enterprise Resource Planning (ERP) systems have been developed to facilitate workflow planning. However, cross-organisational workflow planning remains a rather manual process as the current techniques and tools often lack domain-specific knowledge to support automation in workflow planning and updates. Moreover, the planners may only have limited domain knowledge and do not have a common understanding of all the collaborative workflows, which can be catastrophic, especially in the healthcare domain. Therefore, there is a need for an analysis that over-approximates the cost before any changes in the workflows are implemented. With the cost analysis, the planners can first simulates the changes in the design of workflows, including the task dependencies and resource allocation, and see the effect of the changes in terms of execution time before the changes are implemented in the workflow in practice.

In this paper, we first present a formal modelling language $\mathcal{R}$PL. The language has explicit notions for task dependencies, resource usage and time consumption, which allows the cross-organisational planners to couple various workflows through resources and task dependencies. A preliminary idea of the language is presented in [8]. In addition, we present a technique based on the work in [22] to statically over-approximate the worst execution time of the 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., [15,6]). This enables planners to estimate the effects of the workflows (and its possible changes) in terms of execution time before the actual implementation. 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 introduces the syntax and semantics of the language. Section 3 shows a static analysis to over-approximate the execution time of an $\mathcal{R}$PL program. Section 4 shows the correctness of analysis. Section 5 briefly discusses the related work. Finally, we summarise the paper and discuss possible future work in Section 6.

2 Formal Workflow Modelling Language $\mathcal{R}$PL

In this section, we present a formal modelling language $\mathcal{R}$PL. The language is inspired by an active object language, ABS [20], and has a Java-like syntax and actor-based concurrency model. In an actor-based concurrency model [5], actors are primitives of concurrent computation. They can send a finite number of messages to other actors, spawn a finite number of new actors or modify their

$$
\begin{array}{ll}
P ::= R\ \overline{Cl}\ \{\overline{T\ x};\ s\} & e ::= x \mid g \mid \textbf{this} \\
Cl ::= \textbf{class}\ C\ \{\overline{T\ x};\ \overline{M}\} & g ::= b \mid f? \mid g \wedge g \\
M ::= Sg\ \{\overline{T\ x};\ s\} & s ::= x = rhs \mid \textbf{skip} \mid \textbf{if}\ e\ \{s\} \mid \textbf{wait}(f) \mid \textbf{return}\ e \\
Sg ::= B\ m(\overline{T\ y}) & \quad \mid \textbf{hold}(\overline{r, e}) \mid \textbf{release}(\overline{r, e}) \mid \textbf{cost}(e) \mid s\ ;\ s \\
B ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit} & rhs ::= e \mid \textbf{new}\ C \mid f.\textbf{get} \\
T ::= C \mid B \mid \textbf{Fut}\langle B \rangle & \quad \mid m(x, \overline{e})\ \textbf{after}\ \overline{f?} \mid\ !m(x, \overline{e})\ \textbf{after}\ \overline{f?}
\end{array}
$$

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

private state. A primary feature of the actor-based model is that one message is being processed per actor, preserving the invariants of an actor without locks.

$\mathcal{R}$PL uses explicit notions to express time advancement and to indicate resources required for each task (expressed as a method) and dependencies between tasks. Using cooperative scheduling of method activations, $\mathcal{R}$PL controls the internal interleaving of processes inside an object with explicit scheduling points.

2.1 The syntax of $\mathcal{R}$PL

The syntax of the language 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. Types T in $\mathcal{R}$PL are basic types B, including integer, boolean and unit type, a class C and future types $\textbf{Fut}\langle B \rangle$, which types asynchronous method invocations (see below).

Resources $R : r \mapsto v$ maps resource identifiers r to integer values v, indicating the number of resources r is available. A class declaration $\textbf{class}\ C\ \{\overline{T\ x};\ \overline{M}\}$ 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 $\overline{y}$. 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 $C\ o, \overline{C'\ o'}, \overline{T'\ x}$ where o is always the callee object identifier of the method of class C, $\overline{o'}$ are object identifiers of class $\overline{C'}$ and $\overline{x}$ are the remaining parameters. This assumption is the syntactic sugar that we use to realise the cost analysis introduced in Section 3. Expressions e include guards g, variables x and self-identifier **this**. A guard g allows a process to release control of an object. It can be boolean conditions b, return tests $f?$ checking if the future variable f is resolved, or a conjunction of guards.

Statements include sequential composition, assignment, **if**, **skip**, and **return** are standard. Iterative loops are not included in the language, but can be implemented with recursion. $\mathcal{R}$PL uses $\textbf{hold}(\overline{r, e})$ and $\textbf{release}(\overline{r, e})$ to acquire and return e number of resources r. Statement $\textbf{wait}(f)$ suspends the current process until future f is resolved, while other processes in the same object can be sched-

uled for execution. Statement **cost**(e), the only term in $\mathcal{R}$PL that consumes time, represents e units of time advancement.

The right-hand side *rhs* of an assignment includes expressions e, object creation **new** C, method invocations and synchronisation. Communication in $\mathcal{R}$PL is based on method calls, which can be either synchronous, written as $m(x, \overline{e})$ **after** $\overline{f?}$, or asynchronous, written as $!m(x, \overline{e})$ **after** $\overline{f?}$, where x is the callee object and $\overline{f?}$ is a sequence of futures that must be resolved prior to invoking method m. 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. An asynchronous method invocation is associated to a future variable of type **Fut**$\langle B \rangle$, where B is the return type of the invoked method. Moreover, the expression f.**get** blocks all execution in the object until future f is resolved.

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. The caller object will only be blocked if it tries to retrieve the value of the future with a **get** statement.

Fig. 2 shows a simple program in $\mathcal{R}$PL. The code snippet captures a simple collaboration between the workflows of a retail, a supplier and a courier company. Line 1 models the available resources. Lines 2–10 define a retail sale workflow. First, a request to the supplier for product supply is made asynchronously with associated future f_1 on Line 7. While waiting for the product (until f_1 is resolved), the retailer can continue with other tasks. After getting the product from the supplier (f_1 is resolved), it is sent to the customer by utilising the services of a courier company (Line 8). Lines 11–16 define the deliver workflow of the courier company. A driver and a vehicle (resources) are first acquired to deliver the product (Line 13). Line 14 depicts the time taken for delivery. Afterwards, the acquired resources are released (Line 15). For simplicity, we do not show the implementation of the supply workflow.

```
1  [Driver ↦ 5, Vehicle ↦ 3]
2  class Retail {
3    Unit sale(Retail o, Int ord) {
4      Fut<Bool> f1;
5      Supplier sp = new Supplier;
6      Courier cr = new Courier;
7      f1 = !supply(sp,ord) after;
8      Unit x = deliver(cr,ord,10) after f1?;
9    }
10 }
11 class Courier {
12   Unit deliver(Courier o, Int ord, Int t) {
13     hold(Driver,1)(Vehicle,1);
14     cost(t);
15     release(Driver,1)(Vehicle,1);
16 }}
```

Fig. 2. A simple example.

2.2 The Semantics of $\mathcal{R}$PL

To understand how time advances in $\mathcal{R}$PL and the cost analysis later, 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 with the runtime syntax defined in Fig. 3.

$$
\begin{array}{ll}
cn ::= \varepsilon \mid res \mid obj(o,a,p,q) \mid fut(f,val) & act ::= \varepsilon \mid o \\
\quad \mid invoc(o,f,m,\overline{v}) \mid cn\ cn & val ::= v \mid \bot \\
p ::= \texttt{idle} \mid \{l \mid s\} & res ::= [\ r \mapsto v\ ] \\
q ::= \emptyset \mid \{l \mid s\} \mid q\ \ q & a ::= [\ldots, x \mapsto v, \ldots] \\
s ::= \textsf{cont}(f) \mid \ldots & v ::= o \mid f \mid b \mid k
\end{array}
$$

Fig. 3. Runtime syntax of $\mathcal{R}$PL

A configuration cn includes futures, objects, message invocations, and resources. An empty configuration is ε, and whitespace denotes the associative and commutative union operator on configurations. A future $fut(f, val)$ holds a future identifier f and a return value val, where $\bot$ indicates that future has not been resolved.

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, written as $\{l \mid s\}$, has local variable bindings l and a statement s. A message invocation is a term $invoc(o, f, m, \overline{v})$, where o is a callee object, m a method name, f a future to which method m returns, and $\overline{v}$ the set of actual parameter values for m. Resources res is a mapping from resource identifier r to the number of resources. The statement $\textsf{cont}(f)$ controls the scheduling when a synchronous call completes its execution, returning control to the caller. Values v include object, future identifier, and Boolean, Integer or constant values.

We discuss a selection of the semantics rules of $\mathcal{R}$PL (see Figs. 4 and 5) that are relevant to the analysis later. The rest of the semantics is standard, and can be found in Appendix A. In the semantics, we use the auxiliary functions $dom(l)$ and $dom(a)$ to return the domain of l and a, respectively. The evaluation function $[\![e]\!]_{(a\,o\,l)}$ returns the value of e by computing the expressions and retrieving the value of identifiers stored either in a or l. Moreover, the function $\texttt{atts}(C, o)$ is used to create an object of a class C, which binds `this` to o, and the function $\texttt{bind}(o, f, m, \overline{v}, C)$ 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:

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

The semantics in Figs. 4 and 5 includes object creation, communication, task dependencies, resource management and time advancement. For clarity, we use $\mathbb{F}$ to represent all the futures in the configuration in the semantics.

Rule Wait-False suspends the active process, leaving the object idle if f is not resolved, otherwise Wait-True consumes $\textsf{wait}(f)$. Rule New-Object creates a new object. Rule Get retrieves the value of future f if it is resolved; the reduction on this object is blocked otherwise.

Rules Async-Call and Sync-Call handle the communication between objects through method invocations. To ensure the task dependencies between method calls, the rules first check if all the futures on which the method call depends exists, i.e., if $\overline{f}$ can be found in $\mathbb{F}$ and check if they are resolved. Rule Async-Call creates an invocation message to o' with a fresh unresolved fu-

$$\frac{o' = \texttt{fresh}() \quad a' = \texttt{atts}(C, o')}{obj(o, a, \{l \mid x = \textbf{new}\ C; s\}, q) \rightarrow obj(o, a, \{l \mid x = o'; s\}, q)\ obj(o', a', \texttt{idle}, \emptyset)}\ (\textsc{New-Object})$$

$$\frac{\forall\ f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad o' = [\![e]\!]_{(a \circ l)} \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l \mid x = f'; s\}, q)\ invoc(o', f', m, \overline{v})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Async-Call})$$

$$\frac{v \neq \bot}{obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid x = v; s\}, q)\ fut(f, v)}\ (\textsc{Get})$$

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

$$\frac{v \neq \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid s\}, q)\ fut(f, v)}\ (\textsc{Wait-True})$$

$$\frac{v = \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid \textbf{wait}(f); s\})\ fut(f, v)}\ (\textsc{Wait-False})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o' = [\![e]\!]_{(a \circ l)} \quad o \neq o' \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F} \rightarrow obj(o, a, \{l \mid f' = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; x = f'.\textbf{get}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F}}\ (\textsc{Sync-Call})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad f'' = l(destiny) \quad f' = \texttt{fresh}() \quad \{l' \mid s'\} = \texttt{bind}(o, f', m, \overline{v}, \texttt{class}(o))}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l' \mid s'; \textbf{cont}(f'')\}, q \cup \{l \mid x = f'.\textbf{get}; s\})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Self-Sync-Call})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Async-Call})$$

$$\frac{f'' = l(destiny)}{obj(o, a, \{l' \mid \textbf{cont}(f''), q \cup \{l|s\}) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Sync-Return-Sched})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Sync-Call})$$

$$\frac{[\![e]\!]_{(a \circ l)} = 0}{obj(o, a, \{l \mid \textbf{cost}(e); s\}, q) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Cost})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v \geq 0 \quad where\ v = res(r) - [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{hold}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Hold})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v = res(r) + [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{release}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Release})$$

Fig. 4. A selection of semantics – Part 1

$$\frac{\texttt{strongstable}_{\texttt{t}}(cn)}{cn \to \Phi(cn,t)}\ (\textsc{Tick})$$

where, $\Phi(cn,t) =$

$$\begin{cases} obj(o,a,\{\ l' \mid \textbf{cost}(k);s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{cost}(e);s\},q)\ cn' \\ & \text{and } k = [\![e]\!]_{(a \circ l)} - t \\ obj(o,a,\{l \mid \textbf{hold}(\overline{r,e});s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{hold}(\overline{r,e});s\},q)\ cn' \\ obj(o,a,\{l \mid x = e.\textbf{get};s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid x = e.\textbf{get};s\},q)\ cn' \\ obj(o,a,\texttt{idle},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\texttt{idle},q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

Fig. 5. A selection of semantics – Part 2

ture f', method name m, and actual parameters $\overline{v}$. Rule Self-Sync-Call directly transfers 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-Call specifies a synchronous call to another object, which is replaced by an asynchronous call followed by a **get** statement. In case one of the futures that a synchronous (or asynchronous) method invocations depends on is not yet resolved, the process will be suspended (see Rules (Wait-Async-Call) and (Wait-Sync-Call)). Rules Hold and Release control the resource acquisition and return. Note that it is required to have all the acquired resources to be available in order to consume the **hold** statement; otherwise, the process will be blocked.

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*, no other transition is possible except those evaluating the **cost**(e) statement where e evaluates to some $t \leq 0$, then time advances by the smallest value required to let at least one process execute. To formalize this semantics, we first define stability in Definition 1.

Definition 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. $obj(o,a,\{l \mid x = e.\textbf{get};s\},q)$ *where* $[\![e]\!]_{(a \circ l)} = f$ *and* $fut(f,\bot) \in cn$,
2. $obj(o,a,\{l \mid \textbf{cost}(e);s\},q)$ *where* $[\![e]\!]_{(a \circ l)} \geq t$,
3. $obj(o,a,\{l \mid \textbf{hold}(\overline{r,e});s\},q)$ *with* $res \in cn$,
   where $\exists (r,e) \in \overline{(r,e)}$ *s.t.* $r \in dom(res)$ *and* $res(r) - [\![e]\!]_{(a \circ l)} \leq 0$,
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. $p = \{l \mid x = m(e,\overline{e'})\ \textbf{after}\ \overline{f?};s\}$, *or* $p = \{l \mid x = !m(e,\overline{e'})\ \textbf{after}\ \overline{f?};s\}$,
      where $\exists f \in \overline{f}$ *s.t.* $fut(f,\bot) \in cn$.

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 $[\![e]\!]_{(a,o,l)} = t$. Note that both t-stable and strongly t-stable configurations cannot proceed anymore because every object is stuck either on a $\textbf{cost}(e)$, on unresolved futures, or waiting for some resources. Rule TICK in Fig. 5 handles time advancement when cn is strongly t-stable by advancing time in cn for t units using $\Phi(cn, t)$.

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

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

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 $cn \overset{t_1}{\Longrightarrow} \ldots \overset{t_n}{\Longrightarrow} cn'$; 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 2. *The computational time of* $cn \overset{t_1}{\Longrightarrow} \ldots \overset{t_n}{\Longrightarrow} cn'$ *is* $t_1 + \cdots + t_n$.

The computational time of a configuration cn, written as $\texttt{time}(cn)$, 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 Analysis of $\mathcal{R}$PL program

```
1  [r1 ↦ 2, r2 ↦ 3, r3 ↦ 2]
2  class A {
3    Unit m1(A x, B y, Int k) {
4      Fut<Unit> g1;
5      g1 = !m3(y, k) after;
6      wait(g1);
7      g1.get;}
8    Unit m2(A x, B y, Int k) {
9      Fut<Unit> h1; Unit z;
10     h1 = !m3(y, k) after;
11     z = m1(this, y) after h1?;} }
12 class B {
13   Unit m3(B x, Int k) {
14     hold(r1, 2);
15     cost(k);
16     release(r1, 2);} }
17 {
18   Int k1; Int k2; Int k3;
19   Fut<Unit> f1; Fut<Unit> f2;
20   A a1 = new A; B b1 = new B;
21   cost(k1);
22   f1 = !m2(a1, b1, k3) after;
23   cost(k2);
24   f2 = !m3(b1, k3) after;
25   f1.get;
26   f2.get;}
```

Fig. 6. A running example of an $\mathcal{R}$PL program.

In this section, we describe the cost analysis for an $\mathcal{R}$PL program, which 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. 6 to illustrate the idea of the analysis. Our analysis assumes all $\mathcal{R}$PL programs terminate

and all invoked methods are synchronised. It extends the analysis presented in [22] and to handle a more expressive language with explicit notion of task dependencies and resource allocations.

A cost equation results in a cost expression *exp* that has the following syntax:

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

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 for the finishing time of all of its processes, which gives rise to an upper bound to 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.

3.1 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 [22], 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 $S_m = \texttt{sschem}(\{\{o, \overline{o'}\}\}, s, o)$, which is defined in Definition 3.

Definition 3 (Synchronisation Schema Function). *Let S be a synchronisation schema, s a statement and o a carrier object which is executing s.*

$$\texttt{sschem}(S, s, o) = \begin{cases} S \oplus \{o', \overline{o''}\} & \text{if } s \text{ is } x = m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{f?} \\ & \quad \textit{or, } x = !m(o', \overline{o''}, \overline{e}) \textbf{ after } \overline{f'?} \\ \texttt{sschem}(S, s_1, o) & \text{if } s \text{ is } \textbf{if } e\ \{s_1\} \\ \texttt{sschem}(\texttt{sschem}(S, s', o), s'', o) & \text{if } s \text{ is } s'; s'' \\ S & \text{otherwise.} \end{cases}$$

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 methods defined in Fig. 6 are $S_{m_1} = \{\{x, y\}\},\ S_{m_2} = \{\{x, y\}\},\ S_{m_3} = \{\{x\}\},\ S_{main} = \{\{o_{main}\}, \{a_1, b_1\}\}$.

3.2 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* [22], denoted as $\mathcal{E}$, which is defined as follows:

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

Let o be a carrier object and o' an object that does not belong to the same synchronisation set of o, i.e., $o' \notin S(o)$. 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** $\overline{f?}$, 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 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 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 $\mathcal{E}$ the time 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. 6. The accumulated cost of Line 24 evaluates to $k_1 + max(c_{m_2}, k_2) + c_{m_3}$, which is the cost expression of the *main* method (c_{main}).

Method	*Line*	*Accumulated Cost*	*Method*	*Line*	*Accumulated Cost*
m_1	5	$0 \cdot \langle c_{m_3}, 0 \rangle$	*main*	22	$k_1 \cdot \langle c_{m_2}, 0 \rangle$
m_2	10	$0 \cdot \langle c_{m_3}, 0 \rangle$	*main*	23	$k_1 \cdot \langle c_{m_2}, k_2 \rangle$
m_3	15	k	*main*	24	$k_1 \cdot \langle c_{m_2}, k_2 \rangle \cdot \langle c_{m_3}, 0 \rangle$

3.3 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.

$$
\begin{array}{l}
\boldsymbol{\mathcal{T}}_{S_m}(I, \Psi, o, t_a, t, s) \;= \\
\left\{\begin{array}{ll}
1.\ \boldsymbol{\mathcal{T}}_{S_m}(I', \Psi', o, t'_a, t', s'') & \text{if } s \text{ is } s'; s'', \text{and} \\
 & \quad (I', \Psi', t'_a, t') = \boldsymbol{\mathcal{T}}_{S_m}(I, \Psi, o, t_a, t, s') \\
2.\ (I, \Psi + e, t_a, t + e) & \text{if } s \text{ is } \mathbf{cost}(e) \\
3.\ (I', \Psi', t'_a, t' + c_{m'}) & \text{if } s \text{ is } o = m'(o', \overline{e}) \ \mathbf{after}\ \overline{f?},\ \text{and} \\
 & \quad (I', \Psi', t'_a, t') = \mathbf{trans}_{S_m}(I, \Psi, o, t_a, t, \overline{f}) \\
4.\ (I'[f \mapsto S_m(o)], \Psi', t'_a + c_{m'}, t') & \text{if } s \text{ is } f = !m'(o', \overline{e}) \ \mathbf{after}\ \overline{f'?},\ o' \in S_m(o), \text{and} \\
 & \quad (I', \Psi', t'_a, t') = \mathbf{trans}_{S_m}(I, \Psi, o, t_a, t, \overline{f'}) \\
5.\ (I'[f \mapsto S_m(o')], \Psi'[S_m(o') \mapsto \mathcal{E} \cdot \langle c_{m'}, 0\rangle], t'_a, t') & \\
 & \text{if } s \text{ is } f = m'(o', \overline{e}) \ \mathbf{after}\ \overline{f'?},\ o' \notin S_m(o), \text{and} \\
 & \quad (I', \Psi', t'_a, t') = \mathbf{trans}_{S_m}(I, \Psi, o, t_a, t, \overline{f'}), \text{where} \\
 & \quad \mathcal{E} = \begin{cases} \Psi'(S_m(o')) & \text{if } S_m(o') \in dom(\Psi') \\ t' & \text{otherwise.} \end{cases} \\
6.\ (I', \Psi', t'_a, t') & \text{if } s \text{ is } f.\mathbf{get} \text{ or } \mathbf{wait}(f),\ \text{and} \\
 & \quad (I', \Psi', t'_a, t') = \mathbf{trans}_{S_m}(I, \Psi, o, t_a, t, \{f\}) \\
7.\ (I', \Psi', max(t_a, t_{a_1}), max(t, t_1)) & \text{if } s \text{ is } \mathbf{if}\ e\ \{s_1\}, \text{and} \\
 & \quad (I_1, \Psi_1, t_{a_1}, t_1) = \boldsymbol{\mathcal{T}}_{S_m}(I, \Psi, o, t_a, t, s_1) \\
 & \quad I' = I \cup I_1,\ \text{and} \\
 & \quad \Psi' = \mathtt{upd}(\Psi, \Psi_1, I', dom(I')) \\
8.\ (I, \Psi, t_a, t) & \text{otherwise.}
\end{array}\right.
\end{array}
$$

Fig. 7. The translation function

Given an $\mathcal{R}$PL method m and a synchronisation schema S_m computed based on Section 3.1, 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.

Definition 4 (Translation Environment). *Translation environments, ranged over $\Psi, \Psi', \ldots$, is a mapping from synchronisation sets to their corresponding accumulated costs ($S_m(o) \mapsto \mathcal{E}$).*

Given a synchronisation schema of a method m, S_m, the translation function $\boldsymbol{\mathcal{T}}_{S_m}(I, \Psi, o, t_a, t, s)$ defined in Fig. 7 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 and 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'. We explain in the following the cases of the $\boldsymbol{\mathcal{T}}$ function defined in Fig. 7.

Case 1: Each statement in a sequential composition is translated recursively.

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

Fig. 8. The auxiliary translation function

$$\texttt{upd}(\Psi_1, \Psi_2, I, F) =$$
$$\begin{cases} \Psi_1 & \text{if } F = \emptyset \ \vee \ \Psi_2 = \emptyset \\ \Psi_2 & \text{if } \Psi_1 = \emptyset \\ \texttt{upd}(\Psi_1[I(f) \mapsto max(\Psi_1(I(f)), \Psi_2(I(f)))], \Psi_2, I, F') & \text{if } F = F' \cup f \wedge I(f) \in dom(\Psi_1) \wedge I(f) \in dom(\Psi_2) \\ \texttt{upd}(\Psi_1, \Psi_2, I, F') & \text{if } F = F' \cup f \wedge I(f) \in dom(\Psi_1) \wedge I(f) \notin dom(\Psi_2) \\ \texttt{upd}(\Psi_1[I(f) \mapsto \Psi_2(I(f))], \Psi_2, I, F') & \text{if } F = F' \cup f \wedge I(f) \notin dom(\Psi_1) \wedge I(f) \in dom(\Psi_2) \end{cases}$$

Fig. 9. The auxiliary update function

Case 2: When s is a $\textbf{cost}(e)$ statement, the function updates the current cost t and the accumulated cost Ψ by adding the cost e to them.

Case 3: If s is a synchronous method invocation $m'(o', \overline{e})$ **after** $\overline{f?}$, since the method can only be invoked after the futures $\overline{f}$[1] have been resolved, we need to first compute the cost of all methods associating to $\overline{f?}$ with the auxiliary function $\textbf{trans}_{S_m}(I, \Psi, o, t_a, t, \overline{f})$ in Fig. 8 (see below for explanation). After computing the cost of executing the methods associating to $\overline{f}$, the cost of method m', $c_{m'}$, is added to the accumulated cost t'.

Case 4 & 5: The next two cases corresponds to s as an asynchronous method invocation $!m'(o', \overline{e})$ **after** $\overline{f?}$. Similar to **Case 3**, we first compute the cost of all methods associating to $\overline{f?}$. **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 t'_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 $f.\textbf{get}$ or $\textbf{wait}(f)$ statement, we compute the cost by utilising function $\textbf{trans}_{S_m}(I, \Psi, x, t_a, t, \{f\})$.

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_1}, and the current cost t with the maximum of t and t_1.

[1] We refer $\overline{f}$ to a (possibly empty) set of futures by overloading the overline notation.

The resulting I' is the union of I and I_1. We further update the translation environment with the auxiliary update function defined in Fig. 9.

The trans function. Similar to the translation function $\mathcal{T}$, the auxilary function **trans** in Fig. 8 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 1. We show how the translation function can be applied on the methods defined in Fig. 6. Let $S = \{\{o\}, \{a_1, b_1\}\}$, $S_1 = \{\{x, y\}\}$, $S_2 = \{\{x, y\}\}$ and $S_3 = \{\{x\}\}$ (as computed in Section 3.2). We use s_i to indicate the sequence of statements of a method body starting from line i.

Translation of method m_1 : $\mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, 0, g_1 = !m_3(y, k)\ \textbf{after}; s_6)$
$= \mathcal{T}_{S_1}(\{g_1 \mapsto \{x, y\}\}, \emptyset, x, c_{m_3}, 0, \textbf{wait}(g_1); s_7)$
$= \mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, c_{m_3}, g_1.\textbf{get})$
$= (\emptyset, \emptyset, 0, \boxed{c_{m_3}})$

Translation of method m_2 : $\mathcal{T}_{S_2}(\emptyset, \emptyset, x, 0, 0, h_1 = !m_3(y, k)\ \textbf{after}; s_{11})$
$= \mathcal{T}_{S_2}(\{h_1 \mapsto \{x, y\}\}, \emptyset, x, c_{m_3}, 0, z = m_1(\textbf{this}, y)\ \textbf{after}\ h1?)$
$= (\emptyset, \emptyset, 0, \boxed{c_{m_3} + c_{m_1}})$

Translation of method m_3 : $\mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, 0, \textbf{hold}(r_1, 2); s_{15})$
$= \mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, 0, \textbf{cost}(k); s_{16})$
$= \mathcal{T}_{S_3}(\emptyset, \emptyset, x, 0, k, \textbf{release}(r_1, 2))$
$= (\emptyset, \emptyset, 0, \boxed{k})$

Translation of method *main* :

$$
\begin{aligned}
&\mathcal{T}_S(\emptyset,\emptyset,o,0,0,\texttt{A}\ a_1 = \textbf{new}\ \texttt{A};\ \texttt{B}\ b_1 = \textbf{new}\ \texttt{B};\ s_{21})\\
=\ &\mathcal{T}_S(\emptyset,\emptyset,o,0,0,\textbf{cost}(k_1);\ s_{22})\\
=\ &\mathcal{T}_S(\emptyset,\emptyset,o,0,k_1,f_1 = !m_2(a_1,b_1,k_3)\ \textbf{after};\ s_{23})\\
=\ &\mathcal{T}_S(\{f_1\mapsto\{a_1,b_1\}\},\{\{a_1,b_1\}\mapsto k_1\cdot\langle c_{m_2},0\rangle\},o,0,k_1,\textbf{cost}(k_2);\ s_{24})\\
=\ &\mathcal{T}_S(\{f_1\mapsto\{a_1,b_1\}\},\{\{a_1,b_1\}\mapsto k_1\cdot\langle c_{m_2},k_2\rangle\},o,0,\\
&\qquad\qquad k_1+k_2,f_2 = !m_3(b_1,k_3)\ \textbf{after};\ s_{25})\\
=\ &\mathcal{T}_S(\{f_1\mapsto\{a_1,b_1\},f_2\mapsto\{a_1,b_1\}\},\{\{a_1,b_1\}\mapsto k_1\cdot\langle c_{m_2},k_2\rangle\cdot\langle c_{m_3},0\rangle\},o,0,\\
&\qquad\qquad k_1+k_2,f_1.\textbf{get};\ s_{26})\\
=\ &\mathcal{T}_S(\emptyset,\emptyset,o,0,max(k_1+k_2,k_1\cdot\langle c_{m_2},k_2\rangle\cdot\langle c_{m_3},0\rangle),f_2.\textbf{get})\\
=\ &(\emptyset,\emptyset,0,\ max(k_1+k_2,k_1\cdot\langle c_{m_2},k_2\rangle\cdot\langle c_{m_3},0\rangle)\ )
\end{aligned}
$$

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.

4 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:

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

where m_i are the method names and $1 \le i \le n$, 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 (Cost of Program). *Let* $\mathcal{P} = (R\ \overline{C}\ \{\overline{T\ x};\ s\})$ *be an* $\mathcal{R}$PL *program, where* $\overline{C} = \textbf{class}\ C_1\ \{\overline{T\ x};\ B\ m_1(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\}$

$$\vdots$$

$$\textbf{class}\ C_j\ \{\overline{T\ x};\ B\ m_k(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\ B\ m_n(\overline{T\ y})\{\overline{T'\ x};\ s_n\}\}$$

Then for every $1 \le i \le n$ *and* $1 \le j \le m$, *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 2. The cost program of Fig. 6 is shown as follows, where each cost expression is computed in Example 1.

$$eq_{m_1} = c_{m_3} \ , \quad eq_{m_2} = c_{m_3} + c_{m_1} \ , \quad eq_{m_3} = k \ ,$$
$$eq_{main} = max(k_1 + k_2, k_1 \cdot \langle c_{m_2}, k_2 \rangle \cdot \langle c_{m_3}, 0 \rangle) \ .$$

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 (Sketch). The proof is similar to the one proven in [22]. The main idea is to first extend function $\boldsymbol{\mathcal{T}}$ 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 $\boldsymbol{\mathcal{T}}(cn)$, then $\mathcal{U}(\mathcal{P}) - \texttt{time}(cn \Rightarrow^* cn')$ is a solution of $\boldsymbol{\mathcal{T}}(cn')$.

5 Related Work

Comprehensive research has been performed on modelling business process workflows, such as Business Process Execution Language (BPEL) [23,9], Business Process Model and Notation (BPMN) [25], Petri-nets [1] and Yet Another Workflow Language (YAWL) [3]. BPEL is an executable language for simulating process behaviour, whereas BPMN uses a graphical notation to represent business process descriptions. Petri-nets has been used to formalize both BPEL and BPMN [10,18]. YAWL is inspired by Petri nets, and is a powerful workflow specification language with independent semantics. Different formal approches based on e.g., pi-calculus [4], timed automata [17], CSP [26] have been developed to analyse and reason about models of business process workflows. Compared to our proposed approach, the main focus of these techniques is on intra-organisational workflows and have limited support for coordinating tasks and resources in workflows that are across organisational.

Approaches have been proposed to merge business process models, e.g., [16] presents an approach to merge two business processes based on Event-driven Process Chains [24], which has been implemented in the process mining framework ProM [12], and [21] describes a technique that generates a configurable business process with a pair of business processes as input. To the best of our knowledge, these techniques do not consider connecting workflows across organisations.

Numerous techniques have been introduced for static cost analysis. For example, [7] presents the first approach to the automatic cost analysis of object-oriented bytecode programs, [19] proposes the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. In [22], authors define a concurrent actor language with time. Also, they define a translation function that uses synchronisation sets to compute a cost equation function for each method definition. Compared to this techniques, this paper handles a more expressive language that is sensitive to task dependencies and resource consumption.

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 use 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 enrich the language such that the resource features, e.g., the experience and specialities, can be explicitly specified, and to extend the analysis to handle non-terminating programs. We also plan to develop an approach to associate workflow resources to ontology models. 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 [11], 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 and update workflows modelled in $\mathcal{R}$PL, 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 framework can eventually contribute to automating planning for cross-organisational workflows.

References

1. van der Aalst, W.M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers **8**(01), 21–66 (1998)
2. van der Aalst, W.M.: Loosely coupled interorganizational workflows:: modeling and analyzing workflows crossing organizational boundaries. Information & management **37**(2), 67–75 (2000)
3. van der Aalst, W.M., Ter Hofstede, A.H.: YAWL: yet another workflow language. Information systems **30**(4), 245–275 (2005)
4. Abouzaid, F.: A mapping from pi-calculus into BPEL. Frontiers in artificial intelligence and applications **143**, 235 (2006)
5. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1985)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. Journal of automated reasoning **46**(2), 161–203 (2011)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012), Quantitative Aspects of Programming Languages (QAPL 2010)
8. Ali, M.R., Pun, V.K.I: Towards a resource-aware formal modelling language for workflow planning. In: Intl. Conf. on Model and Data Engineering. Springer (To appear) (2021)

9. Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C.K., Thatte, S., Yendluri, P., Yiu, A.: Web services business process execution language version 2.0. Working Draft. WS-BPEL TC OASIS (2005)
10. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri nets. Queensland Univ. of Technology, Tech. Rep. pp. 1–30 (2007)
11. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Intl. Conf. on Automated Deduction. LNCS, vol. 9195, pp. 517–526. Springer (2015)
12. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H., Weijters, A., van der Aalst, W.M.: The ProM framework: A new era in process mining tool support. In: Intl. Conf. on Application and Theory of Petri Nets. pp. 444–454. Springer (2005)
13. Dourish, P.: 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. pp. 52–60 (2001)
14. Dumas, M., van der Aalst, W.M., Ter Hofstede, A.H.: Process-aware information systems: bridging people and software through process technology. John Wiley & Sons (2005)
15. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Asian Symposium on Programming Languages and Systems. pp. 275–295. Springer (2014)
16. Gottschalk, F., van der Aalst, W.M., Jansen-Vullers, M.H.: Merging event-driven process chains. In: OTM Confederated Intl. Confs. On the Move to Meaningful Internet Systems. pp. 418–426. Springer (2008)
17. Gruhn, V., Laue, R.: Using timed model checking for verifying workflows. Computer Supported Activity Coordination **2005**, 75–88 (2005)
18. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: Intl. Conf. on Business Process Management. pp. 220–235. Springer (2005)
19. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: European Symposium on Programming Languages and Systems. pp. 132–157. Springer (2015)
20. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for Abstract Behavioral Specification. In: Intl. Symposium on Formal Methods for Components and Objects. pp. 142–164. Springer (2010)
21. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.: Merging business process models. In: OTM Confederated Intl. Confs." On the Move to Meaningful Internet Systems". pp. 96–113. Springer (2010)
22. Laneve, C., Lienhardt, M., Pun, K.I, Román-Díez, G.: Time analysis of actor programs. Journal of Logical and Algebraic Methods in Programming **105**, 1–27 (2019)
23. Matjaz Juric, Benny Mathew, P.S.: Business Process Execution Language for Web Services BPEL and BPEL4WS. Packt Publishing (2006)
24. Mendling, J.: Event-driven process chains (epc). In: Metrics for process models, pp. 17–57. Springer (2008)
25. OMG, B.P.M.: Notation (BPMN) Version 2.0 (2011)
26. Wong, P.Y., Gibbons, J.: Property specifications for workflow modelling. Science of Computer Programming **76**(10), 942–967 (2011)
27. Xu, L., Liu, H., Wang, S., Wang, K.: Modelling and analysis techniques for cross-organizational workflow systems. Systems Research and Behavioral Science: The Official Journal of the Intl. Federation for Systems Research **26**(3), 367–389 (2009)

A Semantics of $\mathcal{R}$PL

The full semantics of $\mathcal{R}$PL is given in Figs. 10 and 11. In addition to the rules introduced in Figs. 4 and 5, we have rules Cond-True and Cond-False if handles conditional statements based on the evaluation of expression e. Rule Return puts the return value into the method's associated future. Rule Skip uses a *skip* in the active process. Rule Activate picks a ready for execution process p from the process pool q using the $\texttt{select}(q)$ funtion.

$$\texttt{select}(q) = \begin{cases} \texttt{idle} & \text{if } \texttt{empty}(q) \\ p & \text{if } \exists\ p \in q \text{ and } \texttt{ready}(p) \\ \texttt{idle} & \text{otherwise.} \end{cases}$$

$$\texttt{ready}(p) = \begin{cases} \text{true} & \text{if } p = \textbf{wait}(e) \text{ and } [\![e]\!]_{(a \circ l)} = \text{true} \\ \text{false} & \text{otherwise.} \end{cases}$$

Rules Assign-Local and Assign-Field allot the value of expression e to a variable x in the local variables l or in the objects' field a, respectively. Rule Wait-False suspends the active process, leaving the processor idle if f is not resolved, otherwise Wait-True consumes $\textbf{wait}(f)$. Rule New-Object creates a new object. Rule Get dereferences the future f if it is resolved; otherwise, the reduction on this object is blocked.

Rules Async-Call and Sync-Call handle the communication between objects through method invocations. To ensure the task dependencies between method calls, the rules first check if all the futures the called method depends on exist, i.e., if $\overline{f}$ belong to $\mathbb{F}$ (a set of all futures in the configuration) and the futures must be resolved. Rule Async-Call creates an invocation message to o' with a fresh unresolved future f', the method name m, and actual parameters $\overline{v}$. Rule Self-Sync-Call directly transfers control of the processor from the caller process to the callee. After the execution of callee process is completed, rule Sync-Return-Sched reactivates the caller process. Rule Sync-Call specifies a synchronous call to an other object, captured by an asynchronous call immediately followed by a **get** statement. Rules Hold and Release control the resource acquisition and return. Note that it is required to have all the acquired resources to be available in order to consume the **hold** statement; otherwise, the process will be blocked. In $\mathcal{R}$PL, the unique statement that consumes time is $\textbf{cost}(e)$. Rule Cost specifies a trivial case when e evaluates to 0. When the configuration cn reaches a stable state, no other transition is feasible apart from those evaluating the $\textbf{cost}(e)$ statement then time is advanced by the smallest value required to let at least one process execute. Rule Tick defines the time advancement where $\Phi(cn, t)$ updates configuration cn for time t.

B Type system of $\mathcal{R}$PL

Fig. 12 illustrates the type system of $\mathcal{R}$PL. A typing context Γ maps names to typings, which assigns types T to variables. Expressions of the basic types are

$$\frac{o' = \texttt{fresh}() \quad a' = \texttt{atts}(C, o')}{obj(o, a, \{l \mid x = \textbf{new}\ C; s\}, q) \rightarrow obj(o, a, \{l \mid x = o'; s\}, q)\ obj(o', a', \texttt{idle}, \emptyset)}\ (\textsc{New-Object})$$

$$\frac{\forall\ f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad o' = [\![e]\!]_{(a \circ l)} \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l \mid x = f'; s\}, q)\ invoc(o', f', m, \overline{v})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Async-Call})$$

$$\frac{v \neq \bot}{obj(o, a, \{l \mid x = f.\textbf{get}; s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid x = v; s\}, q)\ fut(f, v)}\ (\textsc{Get})$$

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

$$\frac{v \neq \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \{l \mid s\}, q)\ fut(f, v)}\ (\textsc{Wait-True})$$

$$\frac{v = \bot}{obj(o, a, \{l \mid \textbf{wait}(f); s\}, q)\ fut(f, v) \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid \textbf{wait}(f); s\})\ fut(f, v)}\ (\textsc{Wait-False})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o' = [\![e]\!]_{(a \circ l)} \quad o \neq o' \quad f' = \texttt{fresh}()}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F} \rightarrow obj(o, a, \{l \mid f' = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; x = f'.\textbf{get}; s\}, q)\ obj(o', a', p, q')\ \mathbb{F}}\ (\textsc{Sync-Call})$$

$$\frac{\forall f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v \neq \bot \quad o = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e'}]\!]_{(a \circ l)} \quad f'' = l(destiny) \quad f' = \texttt{fresh}() \quad \{l' \mid s'\} = \texttt{bind}(o, f', m, \overline{v}, \texttt{class}(o))}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \{l' \mid s'; \textbf{cont}(f'')\}, q \cup \{l \mid x = f'.\textbf{get}; s\})\ fut(f', \bot)\ \mathbb{F}}\ (\textsc{Self-Sync-Call})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = !m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Async-Call})$$

$$\frac{f'' = l(destiny)}{obj(o, a, \{l' \mid \textbf{cont}(f''), q \cup \{l|s\}) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Sync-Return-Sched})$$

$$\frac{\exists f \in \overline{f}.fut(f, v) \in \mathbb{F} \wedge v = \bot}{obj(o, a, \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\}, q)\ \mathbb{F} \rightarrow obj(o, a, \texttt{idle}, q \cup \{l \mid x = m(e, \overline{e'})\ \textbf{after}\ \overline{f?}; s\})\ \mathbb{F}}\ (\textsc{Wait-Sync-Call})$$

$$\frac{[\![e]\!]_{(a \circ l)} = 0}{obj(o, a, \{l \mid \textbf{cost}(e); s\}, q) \rightarrow obj(o, a, \{l \mid s\}, q)}\ (\textsc{Cost})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v \geq 0 \quad where\ v = res(r) - [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{hold}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Hold})$$

$$\frac{\forall (r, e) \in \overline{(r, e)}.r \in dom(res) \wedge v = res(r) + [\![e]\!]_{(a \circ l)}}{obj(o, a, \{l \mid \textbf{release}\overline{(r, e)}; s\}, q)\ res \rightarrow obj(o, a, \{l \mid s\}, q)\ res[\overline{r \mapsto v}]}\ (\textsc{Release})$$

Fig. 10. Full semantics of $\mathcal{R}$PL – Part 1

$$\frac{true = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid \textbf{if}\ e\ \{s_1\}s\},q) \rightarrow obj(o,a,\{l \mid s_1; s\},q)}\ (\textsc{Cond-True})$$

$$\frac{false = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid \textbf{if}\ e\ \{s_1\}s\},q) \rightarrow obj(o,a,\{l \mid s\},q)}\ (\textsc{Cond-False})$$

$$\frac{v = [\![e]\!]_{(a \circ l)} \quad f = l(destiny)}{obj(o,a,\{l \mid \textbf{return}\ e; s\},q)\ fut(f,\bot) \rightarrow obj(o,a,\{l \mid s\},q)\ fut(f,v)}\ (\textsc{Return})$$

$$\frac{cn = cn'}{cn\ cn'' \rightarrow cn'\ cn''}\ (\textsc{Context})$$

$$\frac{x \in dom(a) \quad v = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid x = e; s\},q) \rightarrow obj(o,a[x \mapsto v],\{l \mid s\},q)}\ (\textsc{Field-Assign})$$

$$\frac{x \in dom(l) \quad v = [\![e]\!]_{(a \circ l)}}{obj(o,a,\{l \mid x = e; s\},q) \rightarrow obj(o,a,\{l[x \mapsto v] \mid s\},q)}\ (\textsc{Local-Assign})$$

$$\frac{p = \texttt{select}(q)}{obj(o,a,\texttt{idle},q) \rightarrow obj(o,a,p,q \setminus p)}\ (\textsc{Activate})$$

$$obj(o,a,\{l \mid \textbf{skip}; s\},q) \rightarrow obj(o,a,\{l \mid s\},q)\ (\textsc{Skip})$$

$$\frac{\texttt{strongstable}_t(cn)}{cn \rightarrow \Phi(cn,t)}\ (\textsc{Tick})$$

where,

$$\Phi(cn,t) = \begin{cases} obj(o,a,\{\ l' \mid \textbf{cost}(k); s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{cost}(e); s\},q)\ cn' \\ & \text{and } k = [\![e]\!]_{(a \circ l)} - t \\ obj(o,a,\{l \mid \textbf{hold}\overline{(r,e)}; s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid \textbf{hold}\overline{(r,e)}; s\},q)\ cn' \\ obj(o,a,\{l \mid x = e.\textbf{get}; s\},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\{l \mid x = e.\textbf{get}; s\},q)\ cn' \\ obj(o,a,\texttt{idle},q)\ \Phi(cn',t) & \text{if } cn = obj(o,a,\texttt{idle},q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

Fig. 11. Full semantics of $\mathcal{R}$PL – Part 2

type-checked immediately as in the rule T-Bool. By T-Var, a variable is well-typed if stated in Γ. By T-Get, the **get** expression discloses the type of future. By T-Wait, **wait**(e) is well-typed if type of e is Bool. By T-Poll if type of e is a future, then type of return test $e?$ is Bool. Rule T-And disintegrates guards of type Bool. By T-Return, statement **return** e is well-typed if e types to the type of the method's future. Typing rules for skip, composition, assignment, while, and conditional statements are standard.

By T-New-Object, object creation has a type C. By T-SyncCall, a call to a method m has type B if its actual parameters have types $\overline{T}$ and return tests have types **Fut**$\langle B \rangle$. By T-AsyncCall, an asynchronous method call has type **Fut**$\langle B \rangle$ if the corresponding synchronous call has type B.

$$\frac{\Gamma \vdash e \ : \ \texttt{Bool}}{\Gamma \vdash \textsf{wait}(e)}\ (\text{T-Wait}) \qquad \frac{\Gamma \vdash e \ : \ \textsf{Fut}\langle B\rangle}{\Gamma \vdash e? \ : \ \texttt{Bool}}\ (\text{T-Poll}) \qquad \frac{\Gamma \vdash rhs \ : \ \Gamma(x)}{\Gamma \vdash x = rhs}\ (\text{T-Assign}) \qquad \frac{}{\Gamma \vdash b \ : \ \texttt{Bool}}\ (\text{T-Bool})$$

$$\frac{\Gamma \vdash g_1 \ : \ \texttt{Bool} \quad \Gamma \vdash g_2 \ : \ \texttt{Bool}}{\Gamma \vdash g_1 \wedge g_2 \ : \ \texttt{Bool}}\ (\text{T-And}) \qquad \frac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s \ ; \ s'}\ (\text{T-Composition}) \qquad \frac{\Gamma \vdash e \ : \ B \quad \Gamma(destiny) = \textsf{Fut}\langle B\rangle}{\Gamma \vdash \textsf{return}\ e}\ (\text{T-Return})$$

$$\frac{\Gamma \vdash e \ : \ \texttt{Bool} \quad \Gamma \vdash s}{\Gamma \vdash \textsf{if}\ e\ \{s\}}\ (\text{T-Cond}) \qquad \frac{}{\Gamma \vdash \textsf{skip}}\ (\text{T-Skip})$$

$$\frac{}{\Gamma \vdash \textsf{new}\ C \ : \ C}\ (\text{T-New-Object}) \qquad \frac{\forall\ (r,e) \in \overline{(r,e)}.\ \Gamma \vdash r \ : \ \mathbb{R} \ \wedge\ \Gamma \vdash e \ : \ \texttt{Int}}{\Gamma \vdash \textsf{hold}\overline{(r,e)}}\ (\text{T-Hold}) \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x \ : \ T}\ (\text{T-Var})$$

$$\frac{\forall\ (r,e) \in \overline{(r,e)}.\ \Gamma \vdash r : \mathbb{R} \ \wedge\ \Gamma \vdash e \ : \ \texttt{Int}}{\Gamma \vdash \textsf{release}\overline{(r,e)}}\ (\text{T-Release}) \qquad \frac{\Gamma' \ = \ \Gamma[\overline{y} \mapsto \overline{T}, \overline{x} \mapsto \overline{T'}] \quad \Gamma'[destiny \mapsto \textsf{Fut}\langle B\rangle] \vdash s}{\Gamma \vdash B\ m(\overline{T\ y})\{\overline{T'\ x}\ ;\ s\}}\ (\text{T-Method}) \qquad \frac{\Gamma \vdash e \ : \ \textsf{Fut}\langle B\rangle}{\Gamma \vdash e.\textsf{get} \ : \ B}\ (\text{T-Get})$$

$$\frac{\Gamma \vdash e \ : \ \texttt{Int} \quad \Gamma \vdash r \ : \ \mathbb{R}}{\Gamma \vdash [\ r \mapsto e\ ] : \mathbb{R} \mapsto \texttt{Int}}\ (\text{T-Resource}) \qquad \frac{\forall\ f \in \overline{f}.\ \Gamma \vdash f \ : \ \textsf{Fut}\langle B\rangle \quad \Gamma \vdash e \ : \ C \quad \Gamma \vdash \overline{e'} \ : \ \overline{C} \quad \Gamma \vdash \overline{e''} \ : \ \overline{T}}{\Gamma \vdash m(e, \overline{e'}, \overline{e''})\ \textsf{after}\ \overline{f?} \ : \ B}\ (\text{T-Sync-Call})$$

$$\frac{\Gamma \vdash m(e, \overline{e'}, \overline{e''})\ \textsf{after}\ \overline{f?} \ : \ B}{\Gamma \vdash\ !m(e, \overline{e'}, \overline{e''})\ \textsf{after}\ \overline{f?} \ : \ \textsf{Fut}\langle B\rangle}\ (\text{T-Async-Call}) \qquad \frac{\Gamma[\texttt{this} \mapsto C, \texttt{fields}(C)] \vdash \overline{M}}{\Gamma \vdash \textsf{class}\ C\ \{\overline{T'\ x'};\ \overline{M}\}}\ (\text{T-Class})$$

$$\frac{\Gamma \vdash e \ : \ \texttt{Int}}{\Gamma \vdash \textsf{cost}(e)}\ (\text{T-Cost}) \qquad \frac{\Gamma[\overline{x} \mapsto \overline{T}] \vdash s \quad \Gamma \vdash R \quad \forall\ Cl \in \overline{Cl}.\ \Gamma \vdash Cl}{\Gamma \vdash R\ \overline{Cl}\ \{\ \overline{T\ x};\ s\ \}}\ (\text{T-Program})$$

Fig. 12. Type system of $\mathcal{R}$PL

By T-Program, a $\mathcal{R}$pl program is well-typed if its resources, classes, and its main method are well-typed. By T-Resource, a resource has type $\mathbb{R} \mapsto \texttt{Int}$ if resource identifier r has type $\mathbb{R}$ and e has type `Int`. By T-Hold and T-Release, statements $\textbf{hold}(\overline{r, e})$ and $\textbf{release}(\overline{r, e})$ are well-typed in the typing context Γ if all the resource identifiers r have type $\mathbb{R}$ and all expressions e have type `Int`. A class C is well-typed if its methods $\overline{M}$ are well-typed in the typing context Γ extended by the self identifier **this** and fields of the class, by T-Class. Similarly by T-Method, a method declaration is well-typed if method's body is well-typed in the typing context Γ extended by the typing of formal parameters and local variables.

C Subject Reduction

The initial configuration of a well-typed program includes an object, denoted $obj(start, \varepsilon, p, \emptyset)$, where the p is an active process that corresponds to the activation of the program's main block. A run is an order of reductions of an initial configuration based on the semantic rules defined in section 2.2. To prove the correctness of $\mathcal{R}$pl, we need to show that a run from a well-typed initial configuration will keep well-typed configurations. Let $\Gamma \vdash_R cn$ **ok** shows that a configuration cn is well-typed in the typing context Γ. Fig. 13 presents the typing system of runtime configurations of $\mathcal{R}$pl.

$$\frac{\Gamma \vdash_R cn \ \textbf{ok} \quad \Gamma \vdash_R cn' \ \textbf{ok}}{\Gamma \vdash_R cn \ cn' \ \textbf{ok}} \ \text{(T-Configuration)}$$

$$\frac{\Gamma(v) = T \quad \Gamma \vdash_R val : T}{\Gamma \vdash_R T \ v \ val \ \textbf{ok}} \ \text{(T-State)}$$

$$\frac{\Gamma(f) = \textbf{Fut}\langle B \rangle \quad val \neq \bot \Rightarrow \Gamma(val) = B}{\Gamma \vdash_R fut(f, val) \ \textbf{ok}} \ \text{(T-Future)}$$

$$\frac{\Gamma(v) = T \quad \Gamma(r) = \mathbb{R}}{\Gamma \vdash_R [r \mapsto v] \ \textbf{ok}} \ \text{(T-Resource)}$$

$$\frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}] \quad \Gamma' \vdash_R \overline{T \ x \ val} \ \textbf{ok} \quad \Gamma' \vdash_R s \ \textbf{ok}}{\Gamma \vdash_R (\overline{T \ x \ val}, \ s) \ \textbf{ok}} \ \text{(T-Process)}$$

$$\frac{\Gamma \vdash_R q \ \textbf{ok} \quad \Gamma \vdash_R q' \ \textbf{ok}}{\Gamma \vdash_R q \ q' \ \textbf{ok}} \ \text{(T-Process-Queue)}$$

$$\frac{\texttt{fields}(\Gamma(o)) = [\overline{x} \mapsto \overline{T}] \quad \Gamma' = \Gamma[\overline{x} \mapsto \overline{T}] \quad \Gamma' \vdash_R p \ \textbf{ok} \quad \Gamma' \vdash_R q \ \textbf{ok} \quad \Gamma' \vdash_R \overline{T \ x \ val} \ \textbf{ok}}{\Gamma \vdash_R obj(o, \overline{T \ x \ val}, p, q) \ \textbf{ok}} \ \text{(T-Object)}$$

$$\frac{\Gamma(f) = \textbf{Fut}\langle B \rangle \quad \Gamma(\overline{v}) = \overline{B} \quad \texttt{match}(m, \overline{B} \mapsto B, \Gamma(o))}{\Gamma \vdash_R invoc(o, f, m, \overline{v})} \ \text{(T-Invoc)}$$

Fig. 13. Type system of runtime configurations of $\mathcal{R}$pl

Lemma 1 (Type Preservation). *Let Γ be a typing context and σ a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : T$ and $\sigma \vdash e \to \sigma' \vdash e'$, then there is a typing context Γ' such that $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$, and $\Gamma' \vdash e' : T$.*

Proof. The evaluation of an expression e is defined by the small-step reduction relation $\sigma \vdash e \to \sigma \vdash \sigma(e)$. By assumption, $\Gamma \vdash \sigma$ and $\Gamma \vdash e : T$. Since σ is well-typed, $\Gamma \vdash \sigma(e) : \Gamma(e)$, so $\Gamma \vdash \sigma(e) : T$.

It follows from Lemma 1 that given a well-typed expression e and a well-typed substitution σ, then all states in the reduction order from $\sigma \vdash e$ will be well-typed, independent of the order of reductions.

Theorem 2 (Subject Reduction). *If $\Gamma \vdash_R cn$ **ok** and $cn \mapsto cn'$, then there is a Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash_R cn'$ **ok**.*

Proof. The proof is by induction over the application of transition rules. By Lemma 1, the reduction of an expression in a well-typed object ends in a well-typed object. The transition rules employ when these reductions finish, reducing an expression e in the state σ to the ground term $[\![e]\!]_\sigma$.

- Local-Assign and Field-Assign.
  Let $\Gamma \vdash_R obj(o, \overline{T\ x\ v}, \{\overline{T'\ x'\ v'} \mid x = e; s\}, q)$ **ok**. Let $\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}]$. Then $\Gamma' \vdash x = e; s$, so $\Gamma' \vdash e : \Gamma'(x)$. Assume that $v = [\![e]\!]_{(aol)}$, we need to show $\Gamma \vdash_R obj(o, \overline{T\ x\ v}, \{\overline{T'\ x'\ v'}[x \mapsto v] \mid x = e; s\}, q)$ **ok**, which follows from Lemma 1 as $\Gamma' \vdash v : \Gamma'(x)$.
- Cond-True and Cond-False.
  Let $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{if}\ e\ \{s_1\}s\}, q)$ **ok**. By assumption there is a $\Gamma \subseteq \Gamma'$, such that $\Gamma' \vdash e, \Gamma' \vdash s_1$, and $\Gamma' \vdash s$. Consequently, $\Gamma' \vdash s_1; s$, and both rules, Cond-True and Cond-False maintain well-typedness.
- Skip.
  If $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{skip}; s\}, q)$ **ok**, then $\Gamma \vdash_R obj(o, a, \{\ l \mid s\ \}, q)$ **ok**.
- New-Object.
  Let $\Gamma \vdash_R obj(o, a, \{l \mid x = \textbf{new}\ C; s\}, q)$ **ok**. Since $\texttt{fresh}(o')$, let $\Gamma' = \Gamma[o' \mapsto C]$. Certainly, $\Gamma' \vdash_R obj(o, a, \{l \mid x = o'; s\}, q)$ **ok**.
  By assumption, a' is well-typed in o', therefore, $\Gamma' \vdash_R obj(o', a', \texttt{idle}, \emptyset)$ **ok**.
- Return.
  Assume $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{return}\ e; s\}, q)$ **ok** and $\Gamma \vdash_R fut(f, \bot)$ **ok**. Obviously, $\Gamma \vdash_R obj(o, a, \{\ l \mid s\ \}, q)$ **ok**. As $f = l(desting)$ and l is well-typed, we know that $\Gamma(destiny) = \Gamma(f)$. Let $\Gamma(f) = \textbf{Fut}\langle B\rangle$. By rule T-Return, $\Gamma \vdash_R e$ **ok** $: B$ and by Lemma 1, $\Gamma(v) = B$, so $\Gamma \vdash_R fut(f, v)$ **ok**.
- Get.
  By assumption, $\Gamma \vdash_R obj(o, a, \{l \mid x = e.\textbf{get}; s\}, q)$ **ok**, $\Gamma \vdash_R fut(f, v)$ **ok**, and $f = [\![e]\!]_{(aol)}$. Let $\Gamma(f) = \textbf{Fut}\langle B\rangle$. Consequently, $\Gamma \vdash_R e.\textbf{get} : B$ and $\Gamma(v) = B$, so $\Gamma \vdash x = v$, and $\Gamma \vdash_R obj(o, a, \{l \mid x = v; s\}, q)$ **ok**.
- Cost.
  Let $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{cost}(e); s\}, q)$ **ok** and $\overline{v} = [\![\overline{e}]\!]_{(aol)}$. By T-Cost, $\Gamma(e) :$ `Int` and by Lemma 1, $\Gamma \vdash v =$ `Int`, so $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{cost}([\![e]\!]_{(aol)} - 1); s\}, q)$ **ok**, which is immediate.

– Hold.
By assumption, we have $\Gamma \vdash_R obj(o, a, \{l \mid \mathbf{hold}(\overline{r, e}); s\}, q)$ **ok**, $\Gamma \vdash_R res$ **ok** and $\overline{v} = [\![\overline{e}]\!]_{(aol)}$. Obviously, $\Gamma \vdash_R obj(o, a, \{\ l \mid s\ \}, q)$ **ok**. By T-Resource, $\Gamma \vdash \overline{e} : \overline{Int}$ and by Lemma 1, $\Gamma \vdash \overline{v} = \overline{Int}$, so $\Gamma \vdash_R res[\ \overline{r \mapsto v}\ ]$ **ok**. Similarly for Release.
– Self-Sync-Call.
By assumption, $\Gamma \vdash_R obj(o, a, \{l \mid x = m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?}; s\}, q)$ **ok**, $\Gamma \vdash m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?} : B$, $\Gamma \vdash_R \{l' \mid s'\}$ **ok**, and $\mathtt{fresh}(f)$. Let $\Gamma' = \Gamma[f \mapsto \mathbf{Fut}\langle B\rangle]$. Obviously $\Gamma' \vdash \{\ l' \mid s'\ ;\ \mathbf{cont}(f)\}$, $\Gamma' \vdash x = f.\mathbf{get}$, and $\Gamma' \vdash_R fut(f, \bot)$ **ok**.
– Sync-Call
By assumption, $\Gamma \vdash_R obj(o, a, \{l \mid x = m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?}; s\}, q)$ **ok**, $\Gamma \vdash m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?} : B$, and $\mathtt{fresh}(f)$. Let $\Gamma' = \Gamma[f \mapsto \mathbf{Fut}\langle B\rangle]$. Obviously $\Gamma' \vdash f = m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?}; x = f.\mathbf{get}$.
– Async-Call
Let $\Gamma \vdash_R obj(o, a, \{l \mid x = !m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?}; s\}, q)$ **ok**. By assumption, $\Gamma \vdash !m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?} : \mathbf{Fut}\langle B\rangle$ and by T-Assign, $\Gamma(x) = \mathbf{Fut}\langle B\rangle$. Therefore, $\Gamma \vdash e : C$, $\Gamma \vdash \overline{e'} : \overline{C}$ and $\Gamma \vdash \overline{e''} : \overline{T}$. Assume that $o' = [\![\overline{e}]\!]_{(aol)}$ and $\Gamma(o') = C$ for a class C. Let $\Gamma' = \Gamma[f \mapsto \mathbf{Fut}\langle B\rangle]$. Since $\mathtt{fresh}(f)$ and $f \notin dom(\Gamma)$, so if $\Gamma \vdash_R cn$ **ok**, then $\Gamma' \vdash_R cn$ **ok**. Since $\Gamma \vdash !m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?} = \Gamma' \vdash f$, we get $\Gamma' \vdash_R obj(o, a, \{l \mid x = f; s\}, q)$ **ok**. Moreover, $\Gamma' \vdash_R invoc(o', f, m, \overline{v})$ **ok** and $\Gamma' \vdash_R fut(f, \bot)$ **ok**.
– Invoc.
Let $\Gamma \vdash_R obj(o, a, p, q)$ **ok**, $\Gamma \vdash_R invoc(o, f, m, \overline{v})$ **ok** and $C = \Gamma(o)$, so $\Gamma(f) = \mathbf{Fut}\langle B\rangle$ and $\Gamma(\overline{v}) = \overline{T}$. Let $\overline{x}$ be the formal parameters of m in C. Certainly, the auxiliary function $\mathtt{bind}(o, f, m, \overline{v}, C)$ returns a process $\{l[\overline{B\ x\ v}, \mathbf{Fut}\langle B\rangle]\}$ which is well-typed, and it follows that $\Gamma \vdash_R obj(o, a, p, q \cup \{\mathtt{bind}(o, f, m, \overline{v}, C)\})$ **ok**.

Towards a Resource-Aware Formal Modelling Language for Workflow Planning

Preprint · June 2021

CITATIONS
0

READS
218

2 authors:

Muhammad Rizwan Ali
Høgskulen på Vestlandet
11 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Violet Ka I Pun
Høgskulen på Vestlandet
63 PUBLICATIONS **313** CITATIONS
SEE PROFILE

Towards a Resource-Aware Formal Modelling Language for Workflow Planning *

Muhammad Rizwan Ali[1] and Violet Ka I Pun[1,2]

[1] Western Norway University of Applied Sciences, Norway
{mral,vpu}@hvl.no
[2] University of Oslo, Norway
violet@ifi.uio.no

Abstract. Healthcare industry is in a process of digitalising their activities. One of the most crucial ones is the automation of workflow planning. Workflow planning usually requires domain-specific knowledge, which makes it still a rather manual process requiring domain experts. In addition, workflows in healthcare sectors are mainly cross-organisational. Minor changes in the workflow of a collaborative partner may be propagated to other concurrently running workflows, which can result in substantial negative impacts. In this position paper, we present an initial step of an approach towards automating workflow planning. We show how to connect workflows of collaborative workflows through a resource sensitive formal modelling language. We then use an example to show how the language can be used to model cross-organisational workflows, and we discuss how static analyses can be performed on cost approximation and resource scheduling to facilitate planning automation.

Keywords: Healthcare · cross-organisational workflows · resource planning · formal modelling

1 Introduction

Every industry is, one way or another, undergoing the process of digital transformation, including the healthcare industry. Digital transformation activities in healthcare domain includes pathology imagine analysis, remote patient monitoring through digital wearable devices, digital diagnostics, as well as digitalising *workflow planning*.

Workflows are often *cross-organisational*, especially in healthcare industry, which allow organisations to plan their business workflows across departments within the same organisations or even across multiple organisations. Cross-organisational workflows usually consists of multiple concurrent workflows running in different departments within the same organisation, or in different organisations. For instance, the workflows of diagnosing patients in a clinic may

* Partially supported by *Pathology services in the Western Norwegian Health Region – a center for applied digitization* and *SIRIUS – Centre for Scalable Data Access* (www.sirius-labs.no).

involve a workflow of an external laboratory that examine patients' samples, and a workflow of a courier company transporting patients' samples between the laboratory and the clinic.

In addition, workflow planning usually requires domain-specific knowlege to achieve optimal resource allocation and task scheduling, which makes planning cross-organisational workflows particularly challenging. Furthermore, updating workflows is error-prone: one change in a workflow may result in significant changes in other concurrently running workflows, and a minor mistake could lead to substantial negative impacts, in particular in healthcare domain.

Although digitalising and automating *workflow planning* has been widely investigated, and tools like Process-Aware Information Systems (PAIS) [8] and Enterprise Resource Planning (ERP) systems have been developed to facilitate workflow planning, cross-organisational workflow planning remains a largely manual process as the current existing techniques and tools often lack domain-specific knowledge to support automation in workflow planning and updates.

In this position paper, we present a preliminary step towards automating cross-organisational workflow planning. We propose a resource sensitive formal modelling language $\mathcal{R}$PL with explicit notion of task dependency, which can be used to couple collaborative workflows by means of resources and task dependency. The language is inspired by ABS [12], an active object language [4] extending the Actor [3] model of concurrency with asynchronous method calls and synchronisation using futures. The actor-based concurrency model adopted in $\mathcal{R}$PL is used to capture the interactions between workflow activities in practice: for instance, a doctor can continue to perform treatment on a patient while waiting for laboratory results for the diagnosis of another patient.

The rest of the paper is organised as follows: Section 2 introduces the modelling language. Section 3 uses an example to show how the language can be used to model workflows in the healthcare domain. Section 4 briefly discusses the related work. Finally, we summarise the paper and discuss possible future work in Section 5.

2 A Resource-Aware Modelling language

In this section we present a simple modelling language $\mathcal{R}$PL. The language has a Java-like syntax and actor-based concurrency model, and uses *cooperative* scheduling of method activations to explicitly control the internal interleaving of activities inside a concurrent object group, which can be picturised as a processor containing a set of objects.

We use the simple example presented in Fig. 1 to explain the language. The code snippet captures a simple diagnosis workflow in a clinic. While Line 1 models the available

```
1  [Doctor ↦ 5, Nurse ↦ 7]
2  class Clinic() {
3   Pathology lab = new Pathology();
4       ...
5   Unit diagnose(Patient p) {
6       hold((Doctor,1), (Nurse,1));
7       ... // Check the patient
8       Fut<Int> f = lab!examine(p);
9       release((Doctor,1), (Nurse,1));
10      wait f?;
11      Int x = f.get;
12      hold((Doctor,1));
13      this.report(x) after f?; //Write report
14      release((Doctor,1));
15 }}
```

Fig. 1. Diagnosis in a Clinic

$$
\begin{array}{ll}
P ::= R\ \overline{Cl}\ \{\overline{T\ x};\ s\} & e ::= x \mid g \mid \textbf{this} \\
Cl ::= \textbf{class}\ C\ (\overline{T\ x})\ \{\overline{T\ x};\ \overline{M}\} & g ::= b \mid e? \mid g \wedge g \\
M ::= Sg\ \{\overline{T\ x};\ s\} & s ::= x = rhs \mid \textbf{skip} \mid \textbf{if}\ e\ \{s\}\ \textbf{else}\ \{s\} \mid \textbf{while}\ e\ \{s\} \\
Sg ::= T\ m(\overline{T\ x}) & \quad \mid \textbf{return}\ e \mid \textbf{wait}(g) \mid \textbf{hold}\ \overline{(r, e)} \mid \textbf{release}\ \overline{(r, e)} \\
B ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit} \mid C & \quad \mid s\ ;\ s \\
T ::= B \mid F\langle B \rangle & rhs ::= e \mid \textbf{new}\ C(\overline{e}) \mid \textbf{new local}\ C(\overline{e}) \mid e.\textbf{get} \\
 & \quad \mid e.m(\overline{e})\ \textbf{after}\ \overline{e} \mid e!m(\overline{e})\ \textbf{after}\ \overline{e?}
\end{array}
$$

Fig. 2. Syntax of $\mathcal{R}$PL

resources, the rest defines a class Clinic that diagnoses patients. A Doctor and a Nurse (resources) are first acquired on Line 6 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 8. While waiting for the lab to send back the result (Line 10), the resources are released such that they can diagnose the other patients (Line 9). When the result is ready and retrieved (Line 11), a doctor is acquired to write the report (Lines 12–13), and is released (Line 14) afterwards.

Fig. 2 shows the syntax of the core language. A program P consists of resources R, a set of class declaration $\overline{Cl}$, plus a main method body. Resources R is a partial mapping which associates resource identities r to expressions e. Declarations of classes and methods are standard. The language supports basic types B, and *future* type $F\langle B \rangle$, which is similar to any explicit future construct. Statements s include assignments, conditionals, recursion, return, and sequential composition, which are standard. The **hold** and **release** statements handle resource acquisition and release. The **wait** statement releases the control of the processor until the guard g is fulfilled, which can be either a boolean expression or a resolved future. The right-hand side *rhs* of assignments includes guards g, variables x and self-identifier **this** as expressions e, creating new object in remote and local processors, and getting a future value. Objects in $\mathcal{R}$PL can communicate with each other either synchronously with $e.m(\overline{e})$ **after** $\overline{e?}$ or asynchronously $e!m(\overline{e})$ **after** $\overline{e?}$ *after* the futures $\overline{e}$ are resolved.

3 An Illustrative Example

This section presents a simple motivating example to explain how $\mathcal{R}$PL can be employed in modelling cross-organisational workflows in the healthcare domain. Figs. 3–4 depict a cross-organisational workflow of handling patients upon their arrival in BPMN [18], which is consisting of three concurrently executing workflows. Fig. 5 shows the workflow modelled in $\mathcal{R}$PL.

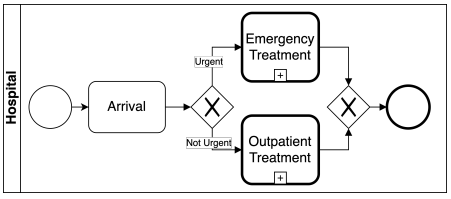

Fig. 3. Workflow of Hospital Admission

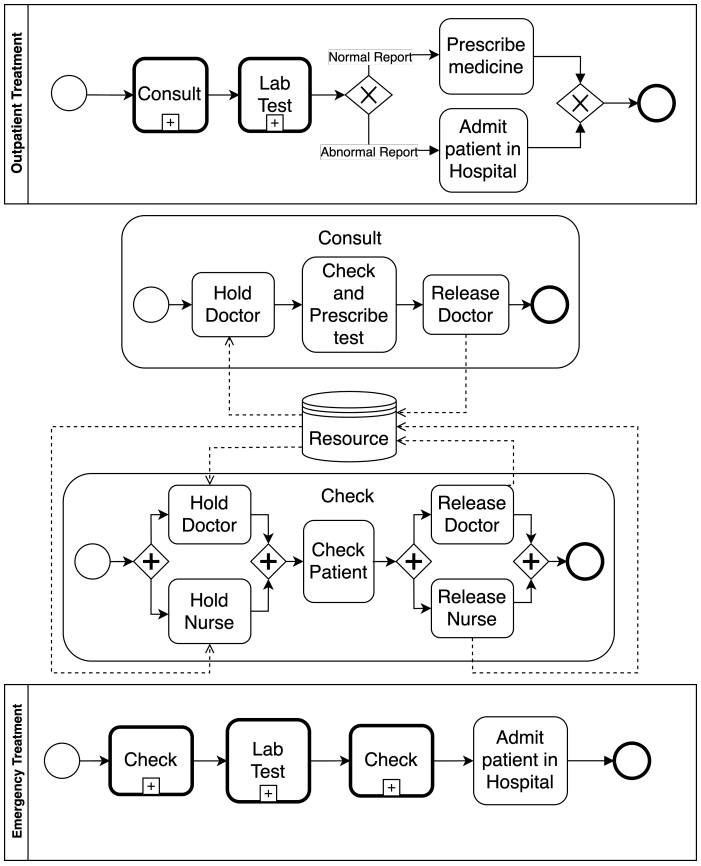

Fig. 4. Workflow of Emergency Department

Fig. 3 shows the hospital admission workflow, which is modelled in $\mathcal{R}$PL by Lines 2–12 in Fig. 5. The workflow connects the ones in Emergency and Outpatient departments. When a patient arrives at the hospital, depending on the his/her emergency status, the patient will be forwarded to either the Emergency department or the Outpatient department.

Fig. 4 illustrates the workflows in the Outpatient and Emergency departments in a hospital that are connected to a database containing the Resource information, which is modelled by Line 1 in $\mathcal{R}$PL in Fig. 5. The first box in Fig. 4 captures the workflow in the Outpatient department that performs Treatments to patients (modelled in $\mathcal{R}$PL by Lines 14–26 in Fig. 5). The Treatment workflow in Fig. 4 shows that doctor Consultation will be first provided to the patient (wrt. Line 20 in Fig. 5). Then, some prescribed tests will be made by the lab (wrt. Line 21 in Fig. 5).[3] Depending on the test result, the patient will either be given a prescription or be admitted to the hospital (wrt. Lines 24–25 in Fig. 5). Consultation is a

[3] For simplicity, we do not show the workflow of test and its implementation.

```
1  [ Doctor ↦ 5 , Nurse ↦ 7]
2  class Hospital() {
3      Pathology lab = new Pathology( );
4      ED e = new ED(lab);
5      OPD o = new OPD(lab);
6      Unit arrival (Bool urgent) {
7          Fut<Int> f;
8          ... //check the status of the patient
9          if (urgent) { f = e!treat() after (); } // Send the patient to emergency department
10         else { f = o!treat() after (); } // Send the patient to outpatient department
11         wait (f?); Int x = f.get;}
12 }
13
14 class OPD(Pathology lab) {
15     Unit consult() { // Consultation
16         hold(Doctor, 1);
17         //Doctor check the patient and prescribe some test
18         release((Doctor, 1)); }
19     Unit treat() {
20         this.consult() after ();
21         Fut<Bool> g1 = lab!test() after ();
22         wait (g1)?;
23         Bool rep = g1.get;
24         if (rep) { ... } // Issue prescription and discharge the patient
25         else { ... }} // Admit patient in hospital
26 }
27
28 class ED(Pathology lab) {
29      Unit check(Int i, Int j) {
30          hold((Doctor, i),(Nurse, j));
31          // checking patient
32          release((Doctor, i),(Nurse, j));}
33      Unit treat() {
34          this.check(1,2) after ();
35          Fut<Bool> h1 = lab!test() after ();
36          wait (h1?);
37          Bool rep = h2.get;
38          this.check(1,1) after h1?;} // Admit patient in Hospital
39 }
```

Fig. 5. Merging Emergency and Outpatient Department Workflow

sub-workflow of Treatment, depicted by the second box in Fig. 4, and modelled in $\mathcal{R}$PL by Lines 15–28 in Fig. 5. In this workflow, one Doctor will be acquired (wrt. Line 16 in Fig. 5) as a resource to check the patient and to prescribe necessary tests, and will be released (wrt. Line 18 in Fig. 5) afterwards.

The workflow of the Emergency department, captured by the last box in Fig. 4, performs Treatments to patients that are in emergency state. Lines 28–39 in Fig. 5 model this in $\mathcal{R}$PL. In this Treatment workflow, a preliminary health Check will be performed on the patient (wrt. Line 34 in Fig. 5). Then, some laboratory tests are made by LabTest (wrt. Line 35 in Fig. 5) followed by a second round of health Check (wrt. Line 38 in Fig. 5). Then, the patient is admitted to the hospital. Health Check is a sub-workflow of Treatment in Emergency department, depicted by the third box in Fig. 4, and modelled in $\mathcal{R}$PL by Lines 29–32 in Fig. 5. In this workflow, one doctor and one nurse are allocated (Line 30 in Fig. 5) as resources to check a patient. They will be released (Line 32 in Fig. 5) at the end of the workflow so that they are available for other patients.

4 Related Work

Extensive research has been conducted in modelling business process workflow, for example, BPEL [16], BPMN [18], (coloured) Petri-Nets [1]. BPEL [16] and BPMN [18] have been developed to specify business processes to model workflow behaviour. While the former is an executable language allowing simulating process behaviour, the latter captures business process descriptions with a graphical notation. Research has been performed on formalising both BPEL as well as BPMN by means of (coloured) Petri-Nets [11,5]. Formal techniques based on e.g., pi-calculus [2,15], timed automata [10], CSP [19,20] have been developed to analyse and reason about models of business process workflows. Compared to our approach, the main focus of these techniques is on intra-organisational workflows and have limited support for coordinating tasks and resources in workflows that are across organisational.

Approaches have been proposed to merge business process models, e.g., [9] presents an approach to merge two business processes based on Event-driven Process Chains (EPCs) [17], which has been implemented in the process mining framework ProM [7], and [14] describes an operator that takes a pair of business processes as input and generates a configurable business process. To the best of our knowledge, these techniques do not consider connecting workflows across organisations.

In [21], the authors present a brief review of intra-organisational modeling techniques and propose two types of architectures, interaction models and routing approaches, to combine inter-organisational workflows. While the former proposes plans for cooperating processes to interact with each other, the latter provides assistance to route cross-organisational workflows in a uniform way.

An automated technique is proposed in [13] to merge collaborating processes in inter-organisational workflows. Instead of connecting those processes with resources and task dependency, the technique is composed of algorithms that connect the nodes of collaborating processes to produce an integrated process.

In contrast to the work discussed above, our work targets cross-organisational workflows involving multiple concurrent workflows from the collaborative partners, where a change in one workflows may have substantial impacts on the others through the resources that connect the workflows or through task dependency.

5 Conclusion

We have presented in this paper a formal language $\mathcal{R}$PL that can be employed to model cross-organisational workflows consisting of concurrently executing workflows. We use an example to show how the language can be used to capture the collaboration between workflows collaborate via resources and task dependency.

This language is the first step towards automating cross-organisational workflow planning. As for the immediate next steps, we plan to enrich the language such that the resource features, e.g., the experience and specialities can be explicitly specified. We also plan to develop a technique to associate workflow resources to ontology models. As for the long-term future work, we plan to develop

analyses to estimate resource allocation and task scheduling, by e.g., constraint solving. Furthermore, we intend to develop verification techniques to ensure the correctness of workflow models in $\mathcal{R}$PL with respect to cross-organisational workflows. A natural starting point is to investigate how to extend KeY-ABS [6], a deductive verification tool for ABS, to support $\mathcal{R}$PL. We foresee that these analyses can eventually contribute to automating planning for cross-organisational workflows.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. J. Circuits Syst. Comput. **8**(1), 21–66 (1998)
2. Abouzaid, F.: A mapping from pi-calculus into BPEL. In: International Conference on Concurrent Engineering. Frontiers in Artificial Intelligence and Applications, vol. 143, pp. 235–242. IOS Press (2006)
3. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
4. de Boer, F., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (Oct 2017)
5. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information and Software Technology **50**(12), 1281–1294 (2008)
6. Din, C.C., Bubel, R., Hähnle, R.: Key-abs: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) International Conference on Automated Deduction. LNCS, vol. 9195, pp. 517–526. Springer (2015)
7. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H., Weijters, A., van Der Aalst, W.M.: The ProM framework: A new era in process mining tool support. In: International Conference on Application and Theory of Petri Nets. pp. 444–454. Springer (2005)
8. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M. (eds.): Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley (2005)
9. Gottschalk, F., van der Aalst, W.M., Jansen-Vullers, M.H.: Merging event-driven process chains. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 418–426. Springer (2008)
10. Gruhn, V., Laue, R.: Using timed model checking for verifying workflows. In: Cordeiro, J., Filipe, J. (eds.) Computer Supported Activity Coordination, Proceedings of the 2nd International Workshop on Computer Supported Activity Coordination. pp. 75–88. INSTICC Press (2005)
11. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to petri nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) International Conference on Business Process Management. vol. 3649, pp. 220–235 (2005)
12. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B., de Boer, F.S., Bonsangue, M.M. (eds.) International Symposium on Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer (2011)

13. Kunchala, J., Yu, J., Yongchareon, S., Liu, C.: An approach to merge collaborating processes of an inter-organizational business process for artifact lifecycle synthesis. Computing **102**(4), 951–976 (2020)
14. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.: Merging business process models. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 96–113. Springer (2010)
15. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. J. Log. Algebraic Methods Program. **70**(1), 96–118 (2007)
16. Matjaz Juric, Benny Mathew, P.S.: Business Process Execution Language for Web Services BPEL and BPEL4WS. Packt Publishing (2006)
17. Mendling, J.: Event-driven process chains (epc). In: Metrics for process models, pp. 17–57. Springer (2008)
18. Object Management Group: Business Process Modeling Notation (BPMN) version 2.0. (2011)
19. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) International Conference on Formal Methods and Software Engineering. LNCS, vol. 5256, pp. 355–374. Springer (2008)
20. Wong, P.Y.H., Gibbons, J.: Property specifications for workflow modelling. Sci. Comput. Program. **76**(10), 942–967 (2011)
21. Xu, L., Liu, H., Wang, S., Wang, K.: Modelling and analysis techniques for cross-organizational workflow systems. Systems Research and Behavioral Science: The Official Journal of the International Federation for Systems Research **26**(3), 367–389 (2009)

PeerJ Computer Science

Petri Net based modeling and analysis for improved resource utilization in cloud computing

Muhammad Rizwan Ali[1], Farooq Ahmad[2], Muhammad Hasanain Chaudary[2], Zuhaib Ashfaq Khan[3], Mohammed A. Alqahtani[4], Jehad Saad Alqurni[5], Zahid Ullah[6] and Wasim Ullah Khan[7]

[1] Department of Computer Science, Western Norway University of Applied Sciences, Bergen, Norway
[2] Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Lahore, Pakistan
[3] Department of Electrical & Computer Engineering, COMSATS University Islamabad, Attock Campus, Attock, Pakistan
[4] Department of Computer Information Systems, College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia
[5] Department of Educational Technology, College of Education, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia
[6] Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia
[7] School of Electrical Engineering and Automation, Wuhan University, Wuhan, China

ABSTRACT

The cloud is a shared pool of systems that provides multiple resources through the Internet, users can access a lot of computing power using their computer. However, with the strong migration rate of multiple applications towards the cloud, more disks and servers are required to store huge data. Most of the cloud storage service providers are replicating full copies of data over multiple data centers to ensure data availability. Further, the replication is not only a costly process but also a wastage of energy resources. Furthermore, erasure codes reduce the storage cost by splitting data in n chunks and storing these chunks into $n + k$ different data centers, to tolerate k failures. Moreover, it also needs extra computation cost to regenerate the data object. Cache-A Replica On Modification (CAROM) is a hybrid file system that gets combined benefits from both the replication and erasure codes to reduce access latency and bandwidth consumption. However, in the literature, no formal analysis of CAROM is available which can validate its performance. To address this issue, this research firstly presents a colored Petri net based formal model of CAROM. The research proceeds by presenting a formal analysis and simulation to validate the performance of the proposed system. This paper contributes towards the utilization of resources in clouds by presenting a comprehensive formal analysis of CAROM.

Subjects Algorithms and Analysis of Algorithms, Computer Education, Data Science
Keywords Cloud computing, Replication, Colored Petri net, Formal analysis

Submitted 30 October 2020
Accepted 7 December 2020
Published 8 February 2021

Corresponding author
Muhammad Hasanain Chaudary, mhchaudary@cuilahore.edu.pk

Academic editor
Muhammad Asif

Additional Information and Declarations can be found on page 19

DOI **10.7717/peerj-cs.351**

INTRODUCTION

Cloud computing is an emerging paradigm of information technology. Moreover, cloud computing is an IT criterion that provides universal access to shared pools of system resources through the Internet. The resources can be provided on demand on pay or in the

How to cite this article Rizwan Ali M, Ahmad F, Hasanain Chaudary M, Ashfaq Khan Z, Alqahtani MA, Saad Alqurni J, Ullah Z, Khan WU. 2021. Petri Net based modeling and analysis for improved resource utilization in cloud computing. **PeerJ Comput. Sci. 7:e351 DOI 10.7717/peerj-cs.351**

form of a subscription. With Internet access growth, cloud computing is emerging in the industry, academia, and society. Due to a large number of resources, the cloud uses virtualization for resource management. Further, clouds need to stimulate data centers' design so that data can be readily available to users anywhere in the world (*Buyya et al., 2009*).

Services

There are four different services in cloud computing.

Software as a Service

Software as a Service (SaaS) is a multi-tenant platform that enables cloud users to deploy their applications to the hosting environment. Further, it supports different cloud applications in a single logical environment to achieve optimization in terms of speed, security, availability, scalability, and economy (*Dillon, Wu & Chang, 2010*).

Platform as a Service

Platform as a Service (PaaS) facilitates the cloud user to organize, develop and manage various applications through a complete "software development lifecycle". Further, it also eliminates the requirement of an organization to traditionally build and maintain the infrastructure, to develop applications (*Sajid & Raza, 2013*). By using SaaS, cloud users can host different applications while PaaS offers a platform to develop different applications (*Dillon, Wu & Chang, 2010*; *Sajid & Raza, 2013*).

Infrastructure as a Service

It offers direct access to resources such as storage, computer, and network resources used for processing (*Dillon, Wu & Chang, 2010*). Infrastructure as a Service (IaaS) sets up an independent virtual machine (VM) to transform the architecture of the application so that multiple copies can be executed on a single machine. Moreover, it provides access to the infrastructure and delivers additional storage for network bandwidth of the corporate web servers and data backups. An important feature of IaaS is that extensive computing can also be switched on, which previously was only accessible to people with the facility of high power computers.

Database as a Service

Database as a Service (DaaS) is a self-service cloud computing model. In DaaS, user request database services and access to the resources. DaaS provides a shared, consolidated program to provide database services on a self-service model (*Mateljan, Čišić & Ogrizović, 2010*).

Deployment models

Based on environmental parameters including openness, storage capacity and proprietorship of the deployment infrastructure, one can choose a deployment model from the types of cloud deployment models given below. The following are the types of cloud computing available in the literature.

Public cloud

Generally, public clouds may be owned and managed by academic or government organizations and it is used by common users and the public. In the traditional regular sense, in public cloud sources, the internet is delivered dynamically and based on self-service via the Internet by an external supplier who shares resources (*Ahmed et al., 2012*). Moreover, security issues occur in such types of clouds and are more prone to attack. That is why the user has access to the public cloud via the correct validations (*Sajid & Raza, 2013*).

Private cloud

Such kind of infrastructure only works for a specific organization while off-premise private cloud is used by one company and the infrastructure is implemented by another company (*Ahmed et al., 2012*). There is no restriction of network bandwidth, security risks, and legal requirements in a private cloud, and data is managed within the organization, which is not permitted in a public cloud (*Kamboj & Ghumman, 2016*).

Hybrid cloud

It is a combination of two or more separate cloud infrastructures (public or private) and forms another type of cloud, the so-called hybrid cloud. This concept is also known as cloud bursting where several integrated cloud infrastructures remain unique entities (*Mell & Grance, 2011*). Hybrid cloud facilitates organizations to shift overflow traffic to the public cloud to prevent service interruption.

Federated cloud

To handle the site failure, cloud infrastructure providers have established different data centers at different geographic locations to ensure reliability. However, this approach has many shortcomings, one problem is that the cloud users may find it difficult to know which remote location is best for their application to host. Cloud service providers have a finite capacity and it is difficult for a cloud infrastructure provider to set up different data centers at different geographic locations. This is why different providers of cloud services fall under one umbrella and form a federated cloud (*Varghese & Buyya, 2018*). In times of work overload, cloud federation offers the opportunity to avail available computational, cost-effective, on-demand, and reliable storage options to other cloud service providers (*Buyya, Ranjan & Calheiros, 2010*). For example, an EU-based EGI federated cloud shares 300 data centers with 20 cloud providers.

Issues

Current data centers are hosting multiple applications having time latency from a few seconds to multiple hours (*Patterson, 2008*). The main focus of Cloud computing is to provide a performance guarantee and to take care of data privacy. With the high growth rate of data on the Cloud, more massive servers' need is rising day by day. Demand for higher performance is being fulfilled by replicating data in multiple data centers worldwide without thinking about energy consumption. Further, on average, every data center utilizes as much energy as 25,000 households. Data centers are costly and unfavorable for the

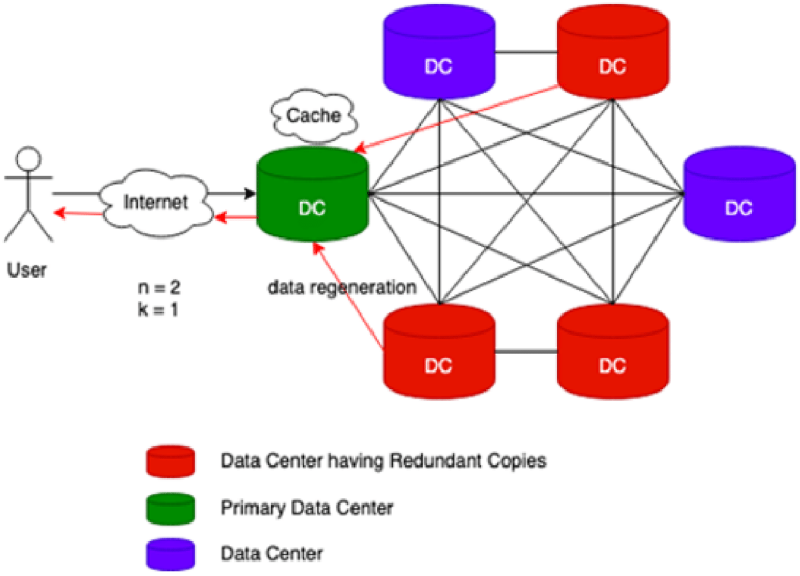

Figure 1 **CAROM process flow.** Full-size DOI: 10.7717/peerj-cs.351/fig-1

environment, as they emit more carbon than both Argentina and the Netherlands (*Patterson, 2008*).

Need of cache-a replica on modification

Cache-A Replica On Modification (CAROM) is a hybrid cloud file system that merges the benefits of both replication and erasure codes. Figure 1 reflects the process flow of CAROM. CAROM has a cache at each data center. Cache points out the local access, and every data center performs as a primary data center. The data object which is frequently accessed is stored in the cache to avoid the extra computational cost. In contrast, those objects that are accessed rarely are divided into m data chunks. Further, distribute them among $n + k$ data nodes, tolerate k failures, and take the storage cost to a minimum and make the data center environment friendly (*Ma et al., 2013*).

Contribution of research

Formal methods are mathematical methods used to model or specify any system. Petri net provides strong mathematical and graphical representations to incorporate concurrency, sequential execution, conflicts, determinism, resource sharing, timing information, communication, synchronization, and distribution in the underlying system. This paper's primary goal is to develop a data scheduling model based on colored Petri net (CPN), which utilizes CAROM to reduce storage cost and bandwidth latency. Statistical analysis is provided to elucidate the performance of the model. Simulation is performed, and verification is also presented of the proposed model.

The rest of the article is organized as follows: "Related Work" presents related work. "Colored Petri Nets" presents basic terminology, notations, and graphical convention about Petri Nets. "Formal Model of CAROM" presents the formal modeling of the CAROM based data scheduling framework. "Simulation" presents a formal analysis of the

developed model. "Analysis" presents the simulations, its results, and the discussion on it. "Conclusion" concludes our work and gives final thoughts about the strengths and weaknesses of our approach.

RELATED WORK

In the cloud, resource scheduling is a challenging field (*Mathew, Sekaran & Jose, 2014*). Magnificent work has been done in resource scheduling in the cloud. Some approaches are relevant to resource scheduling in the cloud. This approach's immediate attention is to optimize time performance, like completion time, total delay, and response time (*Mathew, Sekaran & Jose, 2014*). *Zhan et al. (2015)* provides a detailed survey of cloud computing. Ant colony optimization algorithm for scheduling tasks according to budget is presented in *Zuo et al. (2015)*. In *Adil et al. (2015)*, a heuristic algorithm is proposed for task scheduling. *Kumar & Verma (2012)* presents a genetic algorithm to schedule independent tasks. In *Mateescu, Gentzsch & Ribbens (2011)*, another genetic algorithm is presented that improves the makespan of resources. The authors of *De Assunção, Di Costanzo & Buyya (2010)* proposed an architecture that provides a platform for scientifically, high performance (HPC) applications. The cornerstone of the proposed architecture is the Elastic cluster, which expands the hybrid cloud environment (*De Assunção, Di Costanzo & Buyya, 2010*). Researchers in *Mastelic et al. (2015)* analyzed the assessment between performance and usage costs of various facilities algorithms for using resources from the cloud to expand a cluster capacity. The authors in *Javadi, Abawajy & Buyya (2012)* propose non-disruptive source facilities policies for hybrid cloud environments that they have evaluated using a model-based simulation instead of our real case study performance evaluation. Researchers in *Mattess, Vecchiola & Buyya (2010)* present a facility algorithm for expanding cluster capacity with Amazon EC2 Spot Organizations. Research work in *Yuan et al. (2017)* provides a profit maximization model for private proposals cloud providers using the temporal variation of prices in a hybrid cloud. Although they are similar to many others, they take time, and data and networks' costs are negligible.

However, all the algorithms in the literature were limited to static resources only. With the revolution of cloud computing, the number of data servers is increasing across the world. The construction of the data center is not only cost-effective but also not in favor of the environment. Much focus is given to energy-optimized resource scheduling in cloud computing. The researcher has proposed an aware energy model in the form of directed acyclic graphs in *Gan, Huang & Gao (2010)*. In *Zhao et al. (2016)*, two fitness functions are defined: job completion time and energy.

A researcher in *Shen et al. (2017)* proposed a resource allocation technique that allocates resources to virtual machines taking care of energy. DVFS method has been presented in *Hosseinimotlagh, Khunjush & Samadzadeh (2015)*, which schedules a single task and takes care of the voltage supply. One researcher in *Wu, Chang & Chan (2014)* has presented a virtual machine scheduling algorithm that achieves energy optimization and reduces host temperature. In *Mhedheb et al. (2013)*, a method is presented to reduce both network and server power. Research work in *Xia et al. (2015)* scaled the voltage to

reduce energy costs. Scaled processor utilization and resource consolidation has been presented in *Lee & Zomaya (2012)* for energy optimization.

All these methods focus on reducing the cost of energy without the care of job completion time. In *Beloglazov, Abawajy & Buyya (2012)*, the researcher proposed energy-aware mapping of VMs to cloud servers as a problem with bin-packing, independent of the types of workload. *Klein et al. (2014)* presented a framework of brownout for energy optimization. All users have to bear either time latency or cost on a cloud file system.

COLORED PETRI NETS

Petri nets are bipartite directed graphs with the power of behavioral analysis of the modeled system through it. CPN is a mathematical technique used for modeling parallel systems and graphical analysis of their characteristics (*Jensen, 2013*; *Milner, 1997*; *Ullman, 1998*). CPN is the combination of Petri Net and Standard ML (*Emerson & Sistla, 1996*; *Virendra et al., 2005*). CPN allows defining some user-defined data types along with some standard declarations. It is a general-purpose modeling language and has the power to model parallel systems and analyze their performance. Formal Definition of CPN is presented below (*Jensen & Kristensen, 2009*):

A *net* is a tuple $N = (P, T, A, \Sigma, C, N, E, G, I)$ where:

P is a set of *places*.

T is a set of *transitions*.

A is a set of *arcs* where $P \cup T = P \cap A = T \cap A = \emptyset$

Σ is a set of color sets

C is a color function, that is, $C: P \rightarrow \Sigma$

N is a node function. It maps A into $(P \times T) \cup (T \times P)$.

E is an arc expression function. It maps each arc $a \in A$ into the expression e.

G is a guard function. It maps each transition $t \in T$ to a guard expression g. The output of the guard expression should evaluate to Boolean value: true or false.

I is an initialization function. It maps each place p into an initialization expression i.

We can map each place into a multi-set of tokens in CPN through a mapping function called Marking. Initial Marking reflects the initial state of a model. Final Marking represents the final state of the system.

FORMAL MODEL OF CAROM

For modeling, high-level architecture and the components of the system are identified in the first phase. After that, the identified components' interaction points are defined for the smooth implementation of the component-based architecture. Further, a mixture of top-down and bottom-up approaches is adopted in this paper to model the framework. CAROM uses some part of the local storage disk as a cache. Whenever a written request of a new file is received, the complete file is stored in the reserved memory of each DC named as cache.Whenever the cache is near to be filled, the file least recently used is removed from the cache. It is distributed on $n + k$ data nodes after dividing into n chunks.

However, suppose a read request for a file is received. In that case, it is checked first in the nearest DC. If it is found, then it is downloaded directly, without any computational

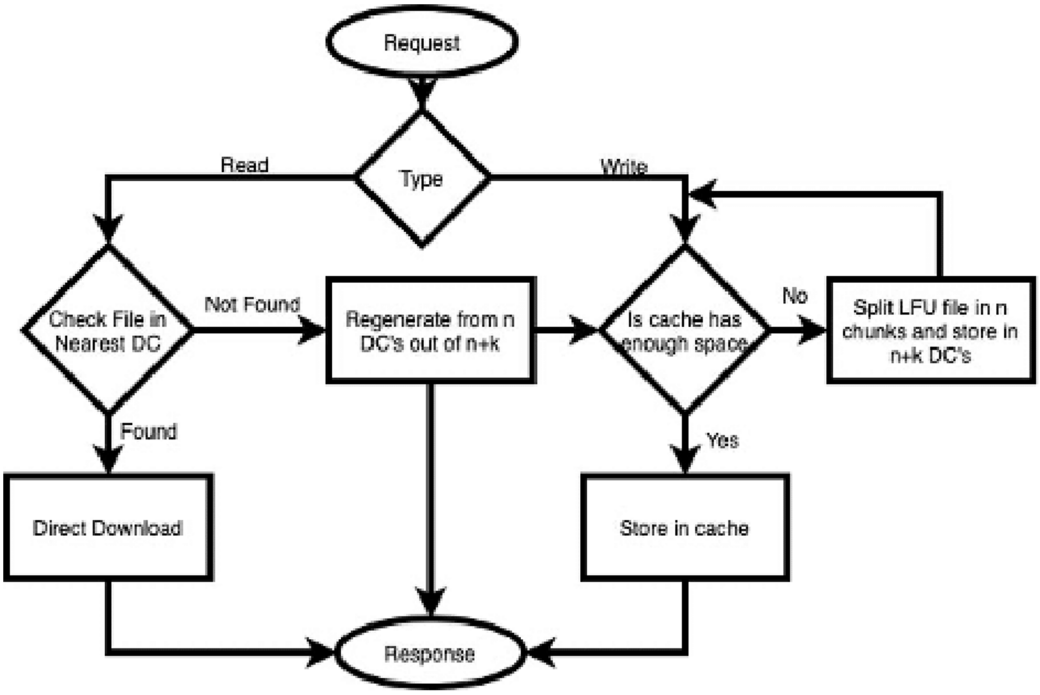

Figure 2 **File access process flow of CAROM.** Full-size DOI: 10.7717/peerj-cs.351/fig-2

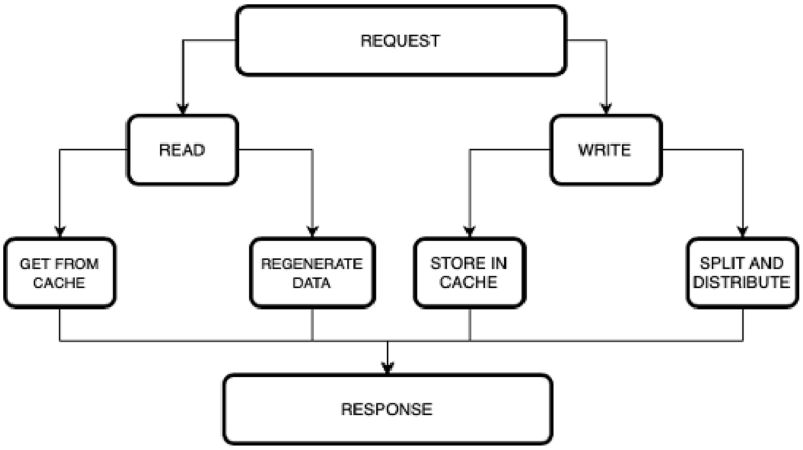

Figure 3 **Hierarchical view of the model.** Full-size DOI: 10.7717/peerj-cs.351/fig-3

cost. Whenever a request of that file is received that is not available in the cache. Data is regenerated from n data nodes out of $n + k$ (*Ma et al., 2013*). The strategy discussed above is presented in the form of a flow chart (see Fig. 2).

Hierarchical view of model

Figure 3 depicts the hierarchical view of the model.

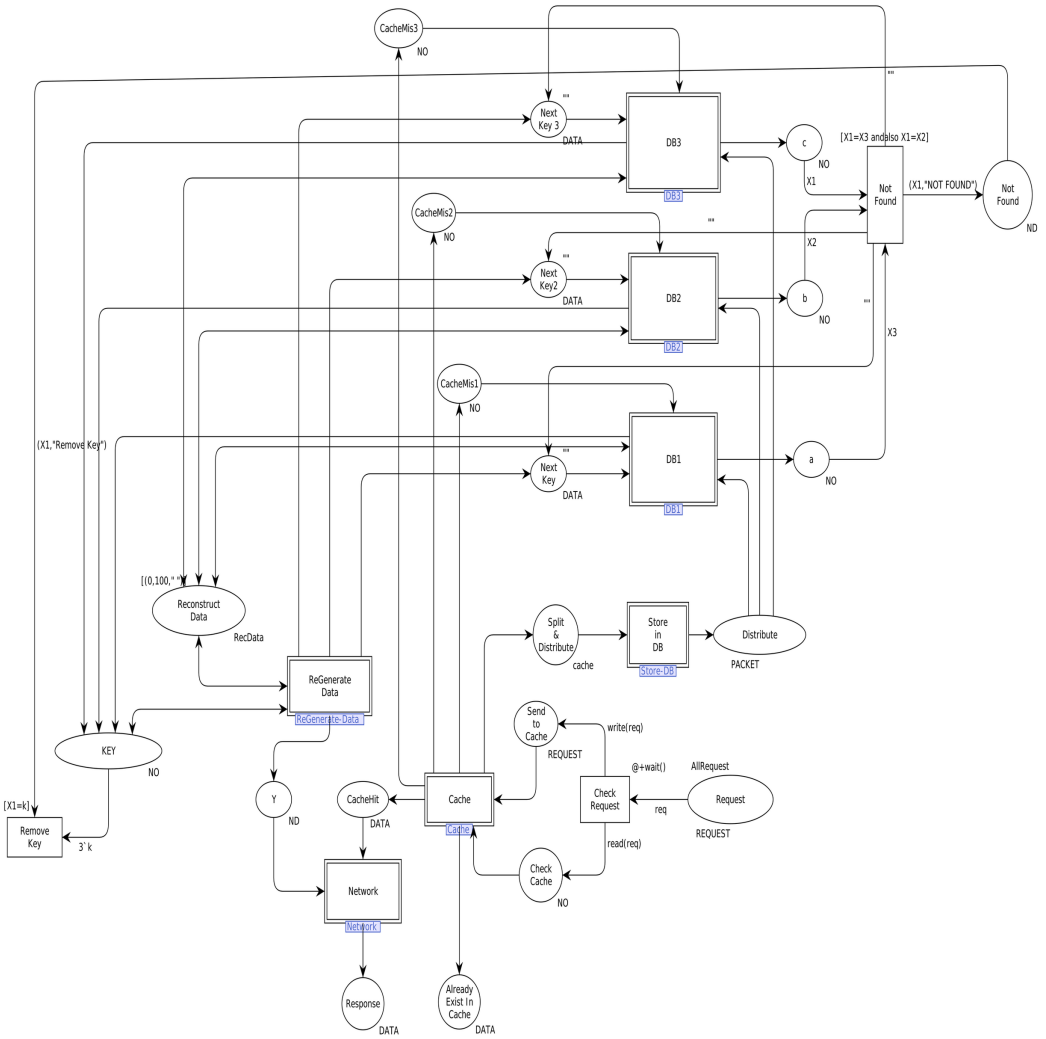

Figure 4 **Top level view of proposed scheme.** Full-size DOI: 10.7717/peerj-cs.351/fig-4

Colored Petri nets model

In order to model the CAROM based framework using CPN, the components *Sender*, *Data Center*, and *Receiver* are developed. *The Data Center* component is further extended to *Cache* and *DataNode* sub-components, as shown in Fig. 3. Table 1 represents the color sets used in the model. As data types, the color sets are mapped to the places of the model given in Fig. 4. For instance, color set *NO*, in the third row of Table 1, is mapped to the place *KEY* while color set *DATA*, in the fourth row of Table 1, is mapped to the place *Next_Key* in the CPN model shown in Fig. 4. Moreover, product type color sets are constructed by taking the cartesian product of the color sets. For instance, the color set *REQUEST* in Table 1 is constructed using color sets *NO, DATA, OP* and *NO*.

Table 2 represents the list of variables used in the model. A variable v is used in the arc inscription, and $Type[v] \in \Sigma$, to fetch the data from the place. Further, the variables construct arc expression, which is assigned to arc a through arc expression function

Table 1 Color sets of the model.

Color set	Defination
colset UNIT = unit;	Unit color set
colset BOOL = bool;	Boolean color set
colset toa = int; closet NO = int;	Integer color sets
colset DATA = string timed;	Timed string color set
colset OP = string timed;	Timed string color set
colset REQUEST = product NO × DATA × OP × NO timed;	Timed product of color set NO of type int, color set DATA of type string, color set OP of type string and color set NO of type int.
colset File= product NO × DATA × toa timed;	Timed product of color set NO of type int, color set DATA of type string and color set toa of type int.
colset ND=product NO × DATA timed;	Timed product of color set NO of type int and color set DATA of type string.
colset RR=product NO × NO timed;	Timed product of color set NO of type int and color set NO of type int.
colset RRL=list RR timed;	Timed list of color set RR.
colset nkd=product NO × NO × DATA;	Product of color set NO of type int, color set NO of type int and color set DATA of type string.
colset RecData = list nkd timed;	Timed list of color set nkd.
colset PACKET = union Data:nkd timed ;	Union of type Data(int*int*string),
colset sendData= product NO × NO × DATA timed;	Timed product of color set NO of type int, color set NO of type int and color set DATA of type string.
colset sendList=list sendData timed;	Timed list of color set sendData.
colset cache = product NO × DATA × NO timed;	Timed product of color set NO of type int, color set DATA of type string and color set NO of type int.
colset CacheList = list cache timed;	Timed list of color set cache.
colset CacheHit= product NO × CacheList timed;	Timed product of a color set NO of type int and color set CacheHit of type list.
colset rcvSplit = product NO × PACKET;	Product of color set NO of type int and color set PACKET of type union.
colset packet=list PACKET timed;	Timed list of color set PACKET of type union.

Table 2 Variables of the model.

Variable	Defination
var p:PACKET;	Variables of colour set PACKET.
var pak:packet;	Variable of colour set packet.
var d,data,next:DATA;	Variables of colour set DATA.
var n,n1,id,k,k1,X1,X2,X3:NO;	Variables of colour set NO.
var cl:CacheList;	Variables of colour set CacheList.
var sl:SendList;	Variable of colour set SendList.
var c:cache;	Variable of colour set cache.
var rd,sd:RecData;	Variables of colour set RecData.
var req,e:REQUEST;	Variables of colour set REQUEST.

E: $A \rightarrow EXPRV$ while $Type[E(a)] = C(p)MS$ where $EXPR$ is the set of expressions and MS is a multiset. A marking is a function M that maps each place $p \in P$ into a multiset of tokens, that is, $M(p) \in C(p)MS$. Table 3 shows values (tokens) to represent the initial marking. Arc expressions are evaluated by assigning the values to the variables in the expressions.

Table 3 Initializations of the model.

Initial marking
val db1=1`[Data(1,1,"C"),Data(2,1,"O"),Data(3,1,"E"),Data(4,1,"P")]@0;
val db2=1`[Data(1,2,"O"), Data(2,2,"U"),Data(3,2,"D"),Data(4,2,"E")]@0;
val db3=1`[Data(1,3,"L"),Data(2,3,"R"),Data(3,3," "), Data(4,3,"T")]@0;
val AllRequest= 1`(1,"COL","WRITE",0)@0 +++ 1`(2,"OUR","WRITE",0)@0+++ 1`(3,"ED ","WRITE",0)@0+++ 1`(1," ","READ",0)@0+++ 1`(2," ","READ",0) @0+++ 2`(4,"PET","WRITE",0)@0+++ 1`(5,"RI ","WRITE",0)@0+++ 1`(5," ","READ",0)@0;

Further, expressions can be converted into functions to be mapped to arcs. Table 4 represents the functions used in this model.

Main module

We first identified high-level components of the system, and then each component is step-wise refined. For such a purpose, hierarchical colored Petri nets are appropriate formalism to make the model more straightforward and understandable. Figure 4 depicts the top-level view of the model. This is a hierarchical model in which multiple substitution transitions connect with places. A substitution transition has its own definition. Therefore, groups are identified from the detailed Petri net model and converted into substitution transitions. There are twenty places and ten transitions, including seven substitution transitions, named *Cache, Store-DB, DB1, DB2, DB3, ReGenerate-Data* and *Receiver*.

Cache module

This module aims to decide whether the data will be directly available from cache or reconstruct it from n different data centers. Figure 5 shows the CPN cache module, and it has ten places and four transitions. Two places are in-sockets and six are out-sockets. Whenever a token is added in the place "*Check Cache*" with operation value "READ", it is sent to transition "*Cache* Checked", which also receives a "cacheList" from the place "*Cache*". Function *member* is a Boolean function. It returns true if the key of token coming from the place "*Check Cache*" is found from *cacheList* (see Table 4 for all declared functions). If *member* function returns true, then the function *retrieve* will get the data against key from the cache.

Further, the function sends data to place "*Cache Hit*" and restores that data object in the cache. In contrast, function *updateLife* will increment the value of the life of this object by 1. On the other side, if the function *member* returns false, then the key is sent to all available data centers through "*CacheMiss*".

Whenever a token is reached in place "*Send to Cache*" with operation value "*WRITE*", it causes enabling of the transition "*Store_in_Cache*" which can only be fired when the cache is not full. Moreover, if the cache is not full and no data object is found with the same key, then token is sent to place "*Cache*" and inserted on the head of the *cacheList*. However, if the cache is full, then the token waits in place "*Send to Cache*" until the function *sort* arranges the *cacheList* with respect to life of data objects. Further, the data object having the least life is removed from cache, and it is sent to place "*Split & Distribute*". If the

Table 4 Functions of the model.

Function	Purpose
fun check(n,k) = if(n>k) then n else k;	Enable transition if first token has greater value
fun wait() = discrete(10,100);	Wait for a random time unit between 10 and 100
fun check1(n,k) = if n=k then k+1 else k;	Increment if both tokens have same value
fun success() = discrete(1,10)<=9;	To check either random number is less than 9
fun success1(n,d) = if success() then 1`(n,d) else empty;	Enable transition if random number is less than 9
fun success2(k) = if success() then 1`k else empty;	Enable transition if random number is less than 9
fun transmit(n,k,d) = if n=k then 1`d else empty;	Enable transition if both tokens have same value
fun transmit1(n,k) = if n>k then 1`k else empty;	Enable transition if first token has greater value
fun read(req:REQUEST) = if #3(req)="READ" then 1`(#1(req)) else empty;	Enable transition with 1st argument of request if 3rd argument of request is "READ"
fun write(req:REQUEST)= if #3(req)="WRITE" then 1`req else empty;	Enable transition with whole request if 3rd argument of request is "WRITE"
fun length [] = 0 | length (h :: t) = 1+length t;	Return length of a list
fun cacheMember(req:REQUEST,[])=false | cacheMember(req,(n,d,k)::t)=if(#1(req))=n then true else cacheMember(req,t);	Check either a file exist in cache or not
fun store(req:REQUEST,cl:CacheList) = ifcacheMember (req,cl) then cl else (#1(req),#2(req),#4(req))::cl;	Store a file on cache
fun member (k1,[]) = false | member (k1,(k2,v2,k3)::t) =if k1=k2 then true else member (k1,t);	Check either a token exist in a list or not
fun member2((k,n,d),[])=false| member2((k,n,d),(k1,n1,d1)::t)=if k=k1 andalso n=n1 andalso d=d1 then true else member2((k,n,d),t);	Check either a token exist in a list or not
fun remDup((k,n,d),sd)= if member2((k,n,d),sd) then sd else (k,n,d)::sd;	Remove duplications
fun insert((k,n,d),[])=[(k,n,d)] | insert((k,n,d),(k1,n1,d1)::t)=if n<=n1 then (k,n,d)::(k1,n1,d1)::t else (k1,n1,d1)::insert((k,n,d),t);	Insert in a list
fun insert1((n,d,k),[])=[(n,d,k)] | insert1((n,d,k),(n1,d1,k1)::t)=if k<=k1 then (n,d,k)::(n1,d1,k1)::t else (n1,d1,k1)::insert1((n,d,k),t);	Insert in a list
fun insert3((k,n,d),[])=[(k,n,d)] | insert3((k,n,d),(k1,n1,d1)::t)=if n<n1 then (k,n,d)::(k1,n1,d1)::t else (k1,n1,d1)::insert3((k,n,d),t);	Insert in a list
fun sort[] = [] | sort ((n,d,k)::t)= insert1((n,d,k),sort t);	Sort with respect to least frequently used
fun sort1[]=[] | sort1((k,n,d)::t) = insert3((k,n,d),sort1 t);	Sort with respect to least frequently used
fun member1(k1,[])=false | member1 (k1,(k,k2,v2)::t)= if k1=k2 then true else member1(k1,t);	Check either a token exist in a list or not
fun recData(k,n1,d,rd)=if member1(n1,rd) then rd else insert((k,n1,d),rd);	Reconstruct data
fun checkDB(k,[])=false | checkDB(k,Data(k1,k2,v)::t) = if k=k1 then true else checkDB(k,t);	Check data in data base
fun recD1(k,Data(k1,k2,v)::t,rd,recData)=if k=k1then recData(k1,k2,v,rd) else recD1(k,t,rd,recData);	Enable transition if data is regenerated
fun found(k,pak,rd,recD1)=if checkDB(k,pak) then recD1(k,pak,rd,recData) else rd;	Enable transition if data is found in data base
fun notFound(k,pak)=if checkDB(k,pak) then empty else 1`k;	Enable transition if data is not found in data base
fun retrieve(k1,[]) = "NOT FOUND" | retrieve (k1,(k2,v2,k3)::t) = if k1=k2 then v2 else retrieve(k1,t);	Retrieve data from data base

(Continued)

Table 4 (continued).

Function	Purpose
fun cacheHit(member,retrieve,k,cl) = if member(k,cl) then 1`retrieve (k,cl) else empty;	Signal to show data is available in cache
fun cacheMis(member,k,cl)= if member(k,cl) then empty else 1`k;	Signal to show data is not available in cache
fun SendTok(k1,k2,k3,k4) = if k3="WRITE" then 3`(k1,k2,k3,k4) else 1`(k1,k2,k3,k4);	Enable either read or write transition
fun updateLife(k,(k1,v1,k2)::t) = if k=k1 then (k1,v1,k2+1) ::t else updateLife(k,t);	Update frequency
fun SplitData(n,d)= let val p1 = packetLength; fun splitdata (n,k,d) = let val d1 = String.size(d) in if d1<=p1 then [(n,k,d)] else (n,k, substring (d,0,p1)):: splitdata(n,k+1,substring(d,p1,d1-p1)) end; in splitdata(n,1,d) end;	Split data
fun Split(k,data) = (List.map(fn (n,n1,d)=>Data(n,n1,d))(SplitData (k,data)));	Split data in $n+k$ chunks

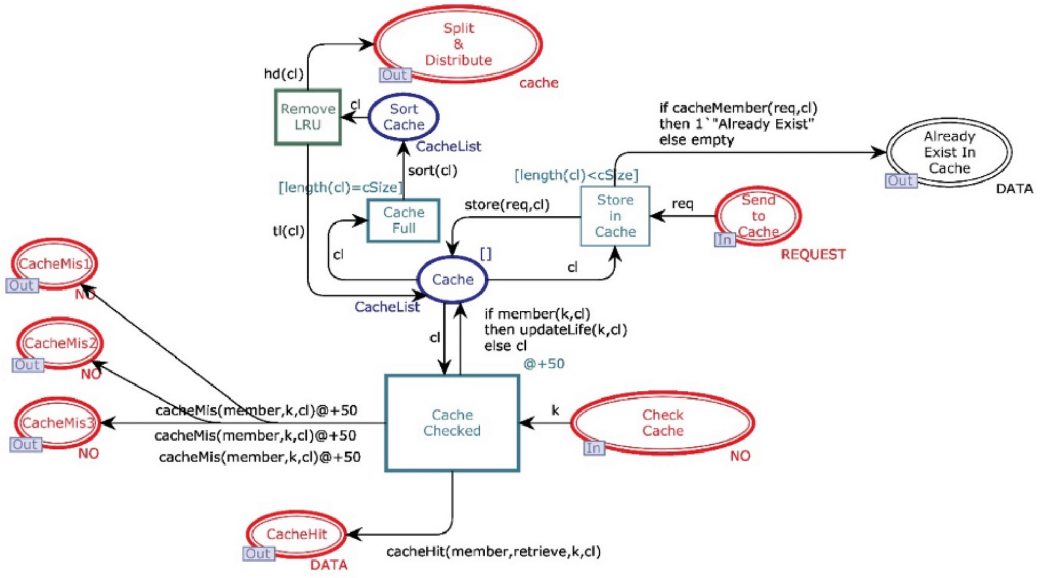

Figure 5 **Cache module.** Full-size DOI: 10.7717/peerj-cs.351/fig-5

cache is not full but the *cacheList* has a record with the same key, then token will be sent to place "*Already Exist In Cache*" by firing the transition "*Send to Cache*".

Store in DB module

Figure 6 shows the Store in DB module of the model. It has two places and one transition. One place is in-socket and one place is out-socket. Whenever the cache is full, the data object with the least life is removed from the cache, and a token is added in the place "Split & Distribute". This token enables the transition "Split Data". Then function Split is

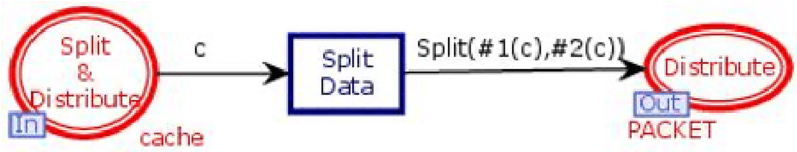

Figure 6 **Store in DB module.** Full-size DOI: 10.7717/peerj-cs.351/fig-6

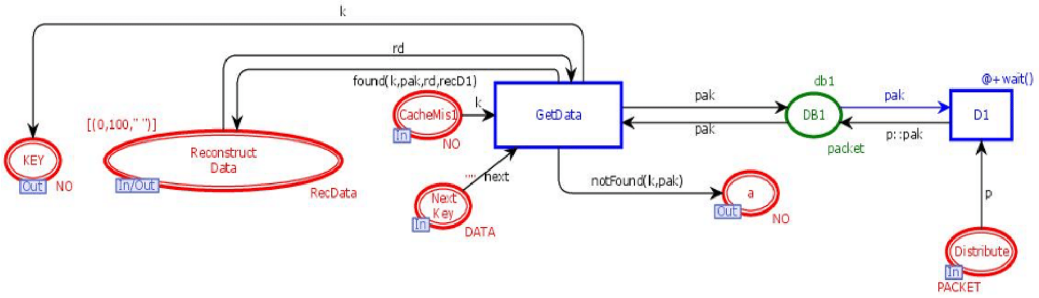

Figure 7 **DB module.** Full-size DOI: 10.7717/peerj-cs.351/fig-7

called, which divides the data value into n data chunks. All the n chunks are sent to place "Distribute" for distribution among $n + k$ databases.

DB module

This module is to retrieve the n data chunks from $n + k$ data centers. DB module contains three in-sockets and two out-sockets. Figure 7 illustrates the DB module of the model. It has seven places and two transitions. Three places are in-sockets, two places are out-sockets and one place is in-out-socket. Whenever a token is reached in place "*Distribute*", it is stored in the database along with its unique key. Whenever a token having a key is added in the place "*CacheMiss*", transition "*GetData*" will check the data chunks against that key. If it is found, then the data chunk and its key will be sent to place "*Reconstruct Data*", which will get n data chunks from $n + k$ data bases to re-generate the original data with the tolerance of k failures.

Regenerate data module

This module is to combine n data chunks to reconstruct data into its original form. Figure 8 shows the ReGenerate-Data module of the model. This module has nine places and four transitions. Two places are in-out-sockets and four are out-sockets. In this module, when we need to reconstruct data the place "*Reconstruct_Data*" receives all data chunks against the search key from all available databases. Transition "*RecD*" remains enable until all data chucks move from place "*Reconstruct_Data*". Then, on arc between the transition "*RecD*" and the place "*Rec*" the function *remDup* (see Table 4) is called, and it removes all the duplications of data chunks. After that, the function *sort1* is called. It sorts data chunks to recontruct the data. The place "*Reconstruct*" holds the token with data in its original form. This place sends data the place "*Reg Data*", which sends the data towards substitution transition "*Receiver*".

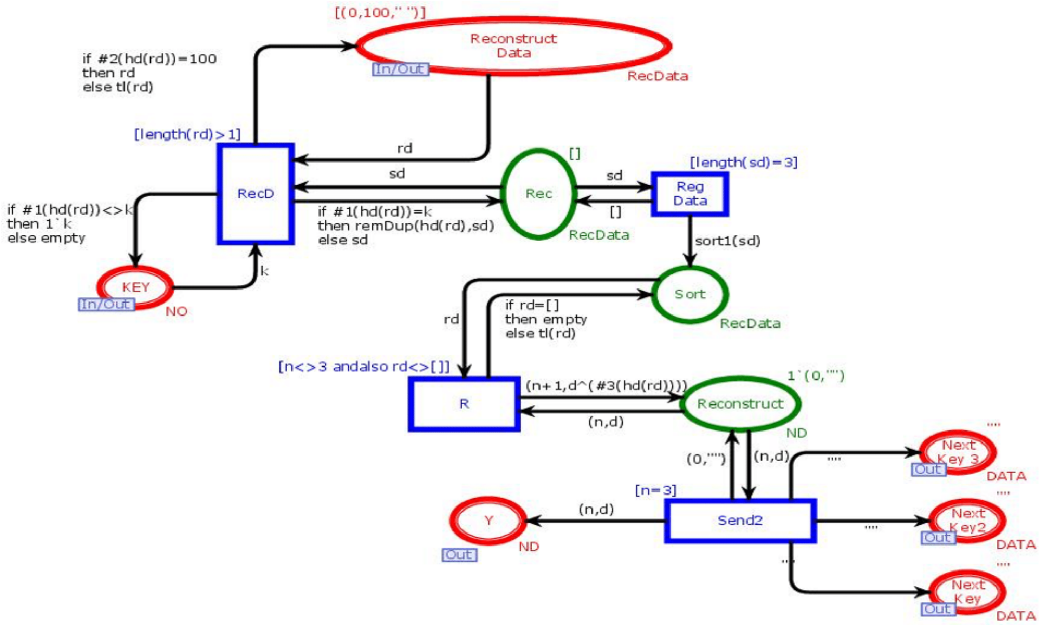

Figure 8 **Regenerate data module.** Full-size DOI: 10.7717/peerj-cs.351/fig-8

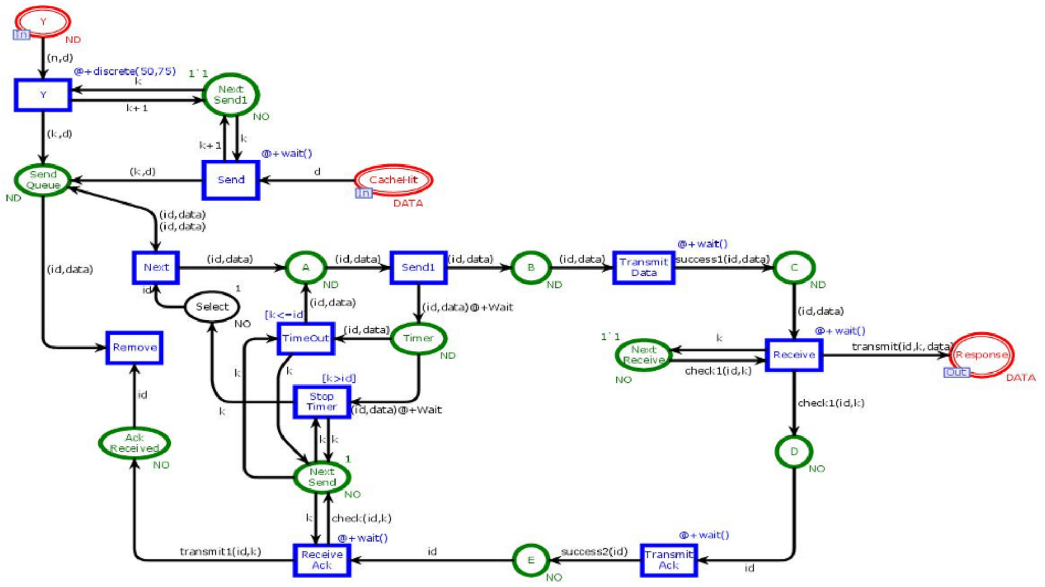

Figure 9 **Receiver module.** Full-size DOI: 10.7717/peerj-cs.351/fig-9

Receiver module

Figure 9 shows the Receiver module. This module is to ensure that data is ultimately transmitted and received by the user. The receiver module has fifteen places and eleven transiotions. Two places are in-sockets and one is out-socket. In this module, whenever a token is reached in Place *"Y"* or *"CacheHit"* it is sent towards the place *"Send Queue"*. When token from the place *"Send Queue"* enables the transition *"Send1"* then chances of

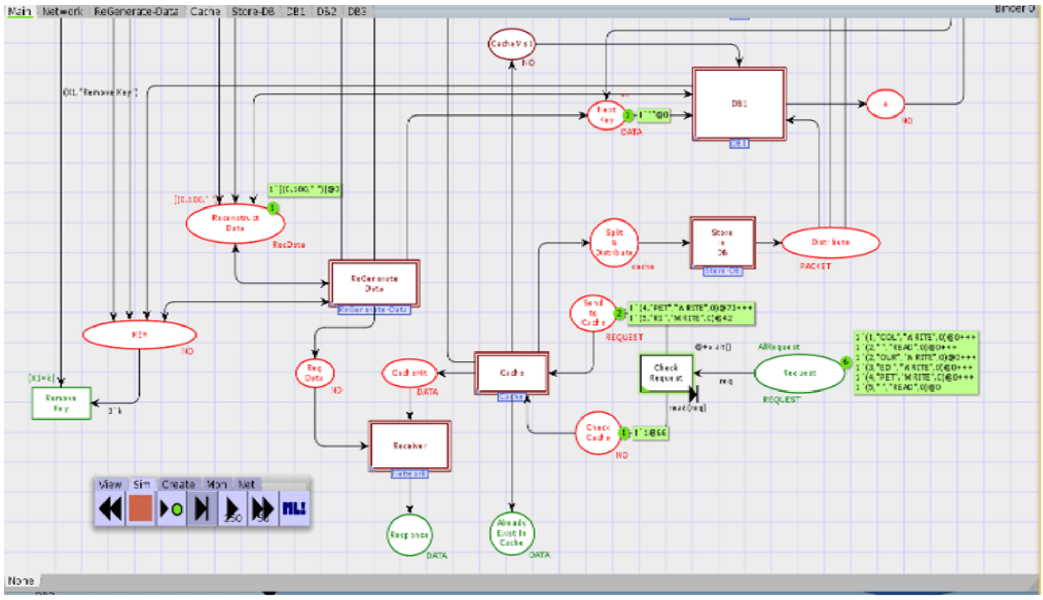

Figure 10 **Partial simulation of the module.** Full-size DOI: 10.7717/peerj-cs.351/fig-10

token lost are 90% over a network. If the token is lost, then place *"Timer"* will receive the token. That token will be sent again to avoid the deadlock situation. If the token is sent to place *C* to enable the transition *"Receiver"* then the transition sends data in place *"Response."* Further, the transition *"TransmitAck"* sends acknowledgment towards the place "*Ack Received"*, which on receiving the token enables the transition *"Remove"* which causes to remove that token from the place *"SendQueue"*.

SIMULATION

Numerous reenactment tools are used to demonstrate and execute a framework, like, process model, SocNetV, Network Workbench. However, it is essential to mention that CPN based formalism supports simulation through CPN Tools. To check the behaviour of the proposed model, we run several manual and ten fully automated simulations of the proposed model with CPN Tools. Figure 10 represents a partial simulation of the model through its intermediate marking (state). In order to get the average completion time of total requests to get both cached and non-cached data, ten simulations are performed (see Table 6). Further, Table 6 shows that simulation 2 gives the high completion time to get cached and non-cached data.

ANALYSIS

To analyze the performance of the proposed model, we performed the following:

Verification of model

State-space analysis of the proposed model is performed to monitor the proposed strategy's possible behavior and amend them accordingly (see Table 5).

Table 5 State space report of the model.

Statistics for O-graph		
State Space		
State Space		
Nodes:	56744	
Arcs:	56746	
Secs:	300	
Status:	56746	
Scc Graph		
Nodes:	56744	
Arcs:	56746	
Secs:	2	
Boundedness Properties		
Best Integer Bounds	Upper	Lower
Cache ′ Cache 1	1	1
DB1 ′ DB1 1	1	1
Main ′ Check_Cache	3	0
Main ′ Request	9	3
Main ′ Send_to_Cache	4	0
Main ′ Response	6	0
Network ′ Next_Receive	1	1
Network ′ Next_Send	1	1
Liveness Properties		
Dead Markings		
409 [543, 2369, 6744, 7430, ...]		
Dead Transition Instances		
None		
Live Transition Instances		
None		
Fairness Properties		
No infinite occurance sequences.		

Performance analysis

To evaluate the performance of the modeled strategy, average delay, throughput, and average queue lengths are collected by performing ten simulations of the model. For such purpose, monitors are applied on the transitions *"Check Request", "Cache Checked", "Split Data", "Get Data", "Reg Data"* and *"Receive"* and *places "CacheHit", "CacheMis1", "Split", "Reconstruct Data"* and *"Response"*. Statistical analysis of output data is performed.

Standard behavioral properties and trace patterns generated by our model are analyzed by state space report. Table 5 illustrates the partial statistics generated by state space with 300 s. It reveals that the occurrence Graph (*O-graph*) has 56,744 nodes and 56,744 arcs. Further, these statistics also depict the boundedness properties. The Upper bound shows the maximum number of tokens in a place, while the lower bound shows the minimum number of tokens that can be added to a specific place. It shows that places

Table 6 Completion time.

Completion time		
	CACHED	NON CACHED
Simulation 1	114	187
Simulation 2	148	205
Simulation 3	98	151
Simulation 4	107	162
Simulation 5	102	132
Simulation 6	99	144
Simulation 7	133	161
Simulation 8	87	142
Simulation 9	127	174
Simulation 10	118	149

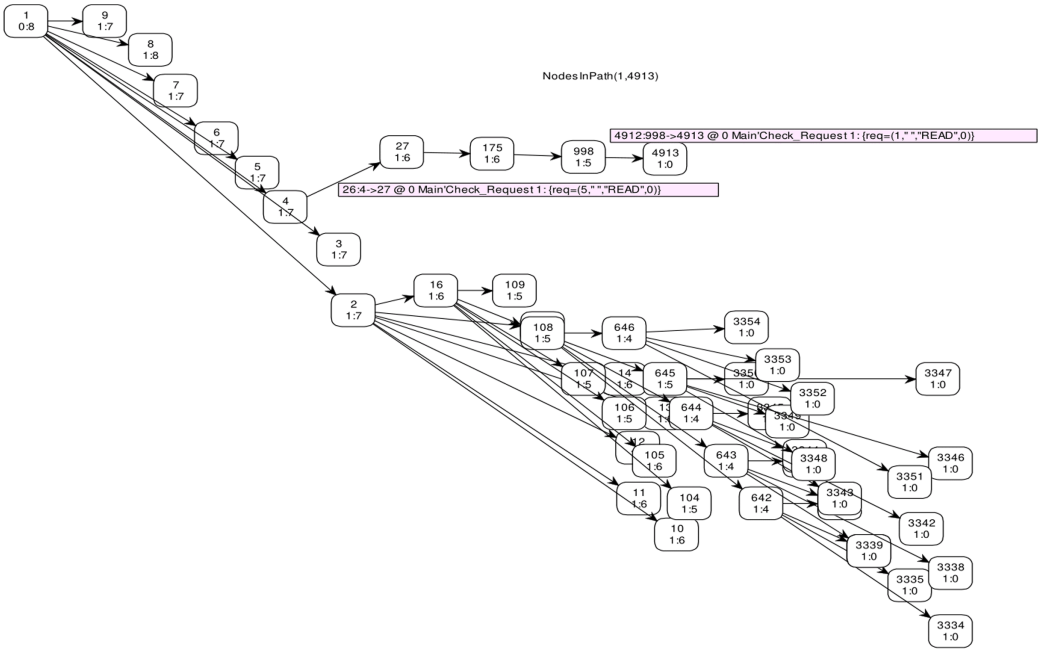

Figure 11 State space graph. Full-size DOI: 10.7717/peerj-cs.351/fig-11

Cache, DB1, Next_Receive and *Next_Send* have both upper and lower bound 1, which means these places always have one token.

However, the upper bound of the place "*Request*" is 9, while it's lower bound is 3. Further, place "*Response*" has upper bound 6 and lower bound equal to zero. It shows that at most 6 requests from place "*Request*" has been fulfilled and stored in place "*Response*". Liveness properties disclose that there exist 409 dead markings. Dead markings are those markings that have no enabled binding elements. Such dead markings are interpreted as final or terminal markings and not deadlock states of the modeled system. The state-space specifies that the model is partially correct and generated results are correct. Therefore, the state-space analysis conveys that the modeled system behaves

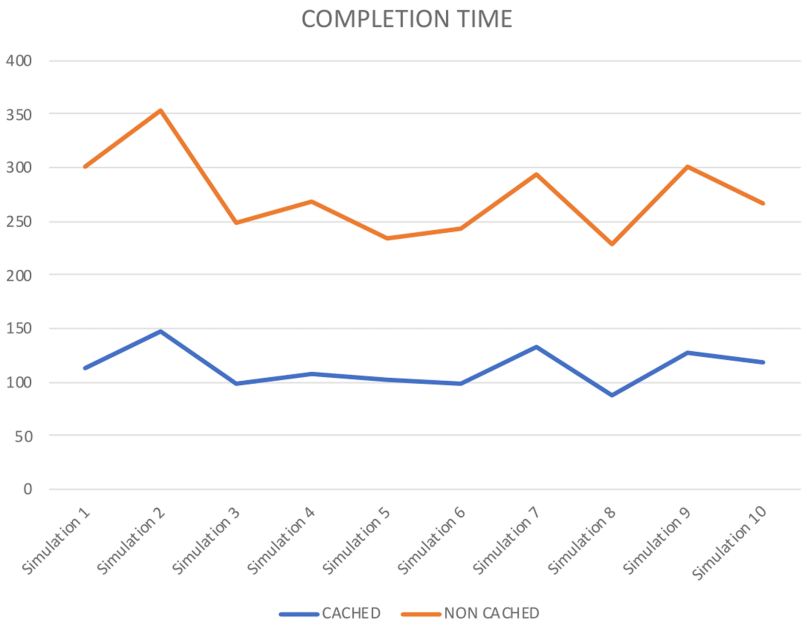

Figure 12 **Completion time.** Full-size DOI: 10.7717/peerj-cs.351/fig-12

according to the requirements and the specifications. Further, the model preserves the properties required for the utilization of storage resources.

The full state space of CPN has 56,744 nodes and 56,744 arcs, which cannot be depicted in the reachability graph. Therefore, Fig. 11 shows a graphical representation of state space from marking M 1–M 4913 by skipping some intermediate markings. In CPN Tools, data collection monitors are applied to compute the average completion time. Table 6 depicts the average completion time of total requests to get both cached and non-cached data for ten simulations.

Figure 12 also represents the completion time for each simulation performed. It shows that in each simulation, cached data takes less time than non-cached. Therefore, it shows that the proposed approach improves storage resource utilization. Further, it validates the precision of our approach.

CONCLUSION

This research is about the issues of data storage and retrieval from cloud-based data centers. Storage cost and bandwidth latency are the two major factors that influence the performance of a system. To reduce the bandwidth latency, most cloud service providers are using multiple copies of data, each on a separate data center across the world. Moreover, data centers are expensive to build and also are unfriendly to the environment. Erasure codes are the techniques that store data of n chunks in $n + k$ data places. However, erasure codes need some extra computation time to regenerate the data. CAROM combined both techniques for dual benefits.

This research formally modeled CAROM using CPN formalism. Furthermore, we formally verified our model with space state analysis. Moreover, we formally analyzed the performance of our model by performing several simulations using monitors in

CPN-Tools. Performance reports generated by CPN-Tools show that the model outperforms the others. In the presented model, the cache size is fixed. The cache is replaced by using the Least Frequently Used replacement algorithm. In the future, we will use some heuristic algorithms to resize and replace the cache in cloud-based systems.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Muhammad Rizwan Ali conceived and designed the experiments, analyzed the data, prepared figures and/or tables, and approved the final draft.
- Farooq Ahmad conceived and designed the experiments, analyzed the data, prepared figures and/or tables, and approved the final draft.
- Muhammad Hasanain Chaudary performed the computation work, prepared figures and/or tables, and approved the final draft.
- Zuhaib Ashfaq Khan conceived and designed the experiments, analyzed the data, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Mohammed A. Alqahtani performed the experiments, performed the computation work, prepared figures and/or tables, and approved the final draft.
- Jehad Saad Alqurni performed the experiments, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Zahid Ullah performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Wasim Ullah Khan performed the experiments, prepared figures and/or tables, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

Raw data are available as a Supplemental File.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.351#supplemental-information.

REFERENCES

Adil SH, Raza K, Ahmed U, Ali SSA, Hashmani M. 2015. Cloud task scheduling using nature inspired meta-heuristic algorithm. In: *2015 International Conference on Open Source Systems & Technologies (ICOSST).* Piscataway: IEEE, 158–164.

Ahmed M, Chowdhury ASMR, Ahmed M, Rafee MMH. 2012. An advanced survey on cloud computing and state-of-the-art research issues. *IJCSI International Journal of Computer Science Issues* **9(1)**:201–207.

Beloglazov A, Abawajy J, Buyya R. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems* **28(5)**:755–768 DOI 10.1016/j.future.2011.04.017.

Buyya R, Ranjan R, Calheiros RN. 2010. Intercloud: utility-oriented federation of cloud computing environments for scaling of application services. In: *International Conference on Algorithms and Architectures for Parallel Processing*, Berlin: Springer, 13–31.

Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. 2009. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* **25(6)**:599–616 DOI 10.1016/j.future.2008.12.001.

De Assunção MD, Di Costanzo A, Buyya R. 2010. A cost-benefit analysis of using cloud computing to extend the capacity of clusters. *Cluster Computing* **13(3)**:335–347 DOI 10.1007/s10586-010-0131-x.

Dillon T, Wu C, Chang E. 2010. Cloud computing: issues and challenges. In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, IEEE, 27–33.

Emerson EA, Sistla AP. 1996. Symmetry and model checking. *Formal Methods in System Design* **9(1)**:105–131 DOI 10.1007/BF00625970.

Gan GN, Huang TL, Gao S. 2010. Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In: *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, IEEE, 60–63.

Hosseinimotlagh S, Khunjush F, Samadzadeh R. 2015. SEATS: smart energy-aware task scheduling in real-time cloud computing. *Journal of Supercomputing* **71(1)**:45–66 DOI 10.1007/s11227-014-1276-9.

Javadi B, Abawajy J, Buyya R. 2012. Failure-aware resource provisioning for hybrid Cloud infrastructure. *Journal of Parallel and Distributed Computing* **72(10)**:1318–1331 DOI 10.1016/j.jpdc.2012.06.012.

Jensen K. 2013. *Coloured Petri nets: basic concepts, analysis methods and practical use.* Vol. 1. Berlin: Springer Science & Business Media.

Jensen K, Kristensen LM. 2009. *Coloured Petri nets: modelling and validation of concurrent systems.* Berlin: Springer Science & Business Media.

Kamboj S, Ghumman NS. 2016. A survey on cloud computing and its types. In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom).* Piscataway: IEEE, 2971–2974.

Klein C, Maggio M, Årzén KE, Hernández-Rodriguez F. 2014. Brownout: Building more robust cloud applications. In: *Proceedings of the 36th International Conference on Software Engineering*, ACM, 700–711.

Kumar P, Verma A. 2012. Independent task scheduling in cloud computing by improved genetic algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering* **2(5)**.

Lee YC, Zomaya AY. 2012. Energy efficient utilization of resources in cloud computing systems. *Journal of Supercomputing* **60(2)**:268–280 DOI 10.1007/s11227-010-0421-3.

Ma Y, Nandagopal T, Puttaswamy KP, Banerjee S. 2013. An ensemble of replication and erasure codes for cloud file systems. In: *INFOCOM, 2013 Proceedings.* Piscataway: IEEE, 1276–1284.

Mastelic T, Oleksiak A, Claussen H, Brandic I, Pierson JM, Vasilakos AV. 2015. Cloud computing: survey on energy efficiency. *ACM Computing Surveys* **47(2)**:33–36 DOI 10.1145/2656204.

Mateescu G, Gentzsch W, Ribbens CJ. 2011. Hybrid computing—where HPC meets grid and cloud computing. *Future Generation Computer Systems* **27(5)**:440–453 DOI 10.1016/j.future.2010.11.003.

Mateljan V, Čišić D, Ogrizović D. 2010. Cloud database-as-a-service (DaaS)-ROI. In: *The 33rd International Convention MIPRO.* Piscataway: IEEE, 1185–1188.

Mathew T, Sekaran KC, Jose J. 2014. Study and analysis of various task scheduling algorithms in the cloud computing environment. In: *ICACCI, 2014 International Conference on Advances in Computing, Communications and Informatics.* Piscataway: IEEE, 658–664.

Mattess M, Vecchiola C, Buyya R. 2010. Managing peak loads by leasing cloud infrastructure services from a spot market. In: *2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC).* Piscataway: IEEE, 180–188.

Mell P, Grance T. 2011. *The NIST definition of cloud computing.* Washington, D.C.: U.S. Department of Commerce.

Mhedheb Y, Jrad F, Tao J, Zhao J, Kołodziej J, Streit A. 2013. Load and thermal-aware VM scheduling on the cloud. In: *International Conference on Algorithms and Architectures for Parallel Processing*, Cham: Springer, 101–114.

Milner R. 1997. *The definition of standard ML: revised.* Cambridge: MIT Press.

Patterson DA. 2008. The data center is the computer. *Communications of the ACM* **51(1)**:105 DOI 10.1145/1327452.1327491.

Sajid M, Raza Z. 2013. Cloud computing: issues & challenges. *International Conference on Cloud, Big Data and Trust* **20(13)**:13–15.

Shen Y, Bao Z, Qin X, Shen J. 2017. Adaptive task scheduling strategy in cloud: when energy consumption meets performance guarantee. *World Wide Web—Internet and Web Information Systems* **20(2)**:155–173.

Ullman JD. 1998. *Elements of ML programming ML97 edition.* Vol. 2. Upper Saddle River: Prentice Hall.

Varghese B, Buyya R. 2018. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems* **79**:849–861 DOI 10.1016/j.future.2017.09.020.

Virendra M, Jadliwala M, Chandrasekaran M, Upadhyaya S. 2005. Quantifying trust in mobile ad-hoc networks. In: *Integration of Knowledge Intensive Multi-Agent Systems, 2005.* Piscataway: IEEE, 65–70.

Wu CM, Chang RS, Chan HY. 2014. A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters. *Future Generation Computer Systems* **37**:141–147 DOI 10.1016/j.future.2013.06.009.

Xia Y, Zhou M, Luo X, Pang S, Zhu Q. 2015. A stochastic approach to analysis of energy-aware DVS-enabled cloud datacenters. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **45(1)**:73–83 DOI 10.1109/TSMC.2014.2331022.

Yuan H, Bi J, Tan W, Li BH. 2017. Temporal task scheduling with constrained service delay for profit maximization in hybrid clouds. *IEEE Transactions on Automation Science and Engineering* **14(1)**:337–348 DOI 10.1109/TASE.2016.2526781.

Zhan ZH, Liu XF, Gong YJ, Zhang J, Chung HSH, Li Y. 2015. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys* **47(4)**:63 DOI 10.1145/2788397.

Zhao Q, Xiong C, Yu C, Zhang C, Zhao X. 2016. A new energy-aware task scheduling method for data-intensive applications in the cloud. *Journal of Network and Computer Applications* **59**:14–27 DOI 10.1016/j.jnca.2015.05.001.

Zuo L, Shu L, Dong S, Zhu C, Hara T. 2015. A multi-objective optimization scheduling method based on the ant colony algorithm in cloud computing. *IEEE Access* **3**:2687–2699 DOI 10.1109/ACCESS.2015.2508940.