Høgskulen på Vestlandet

HVL-rapport nr. 15
2021

Cost Analysis for an Actor-Based Workflow Modelling Language

Muhammad Rizwan Ali and Violet Ka I Pun

Faculty of Engineering and Science
Department of Computer Science, Electrical Engineering and Mathematical Sciences

Høgskulen på Vestlandet
2021

HVL-rapport frå Høgskulen på Vestlandet nr. 15

ISSN 2535-8103
ISBN 978-82-93677-58-1

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

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

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 $\textsf{Fut}\langle B \rangle$, 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 $\textsf{class}\ C\ \{\overline{T\ x};\ \overline{M}\}$ 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 $\textsf{hold}(\overline{r, e})$ and $\textsf{release}(\overline{r, e})$ to acquire and return e number of resources r. Statement $\textsf{wait}(f)$ 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.

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

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

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

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 $\texttt{strongstable}_t(cn)$, 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 $\stackrel{t}{\Rightarrow}$ is $\rightarrow^* \stackrel{t}{\rightarrow} \rightarrow^*$. A computation is $cn \stackrel{t_1}{\Rightarrow} \ldots \stackrel{t_n}{\Rightarrow} cn'$; 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* $cn \stackrel{t_1}{\Rightarrow} \ldots \stackrel{t_n}{\Rightarrow} cn'$ *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 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.

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

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 :
$$\begin{aligned}
&\mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, 0, g_1 = !m_3(y, k)\ \textsf{after}; s_6)\\
&= \mathcal{T}_{S_1}(\{g_1 \mapsto \{x, y\}\}, \emptyset, x, c_{m_3}, 0, \textsf{wait}(g_1); s_7)\\
&= \mathcal{T}_{S_1}(\emptyset, \emptyset, x, 0, c_{m_3}, g_1.\textsf{get})\\
&= (\emptyset, \emptyset, 0, \boxed{c_{m_3}})
\end{aligned}$$

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

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

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 $1 \leq i \leq n$, 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* $\overline{C} = \textbf{class}\ C_1\ \{\overline{T\ x};\ B\ m_1(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\}$
$\vdots$
$\textbf{class}\ C_j\ \{\overline{T\ x};\ B\ m_k(\overline{T\ y})\{\overline{T'\ x};\ s_1\}\ldots\ B\ m_n(\overline{T\ y})\{\overline{T'\ x};\ s_n\}\}$
Then for every $1 \leq i \leq n$ *and* $1 \leq j \leq m$, *let*

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

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

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

Example 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* RPL *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

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

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

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

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

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

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

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

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

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

where,

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

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 $\textbf{hold}(r, e)$ and $\textbf{release}(r, e)$ 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 $v = [\![e]\!]_{(a \circ l)}$, 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 $f = [\![e]\!]_{(a \circ l)}$. 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 $\overline{v} = [\![\overline{e}]\!]_{(a \circ l)}$. By T-Cost, $\Gamma(e) :$ `Int` and by Lemma 1, $\Gamma \vdash v =$ `Int`, so $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{cost}([\![e]\!]_{(a \circ l)} - 1); s\}, q)$ **ok**, which is immediate.

– Hold.
By assumption, we have $\Gamma \vdash_R obj(o, a, \{l \mid \textbf{hold}\overline{(r, e)}; s\}, q)$ **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 $\texttt{fresh}(f)$. Let $\Gamma' = \Gamma[f \mapsto \textbf{Fut}\langle B\rangle]$. Obviously $\Gamma' \vdash \{\ l' \mid s'\ ;\ \textbf{cont}(f)\}$, $\Gamma' \vdash x = f.\textbf{get}$, 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 $\texttt{fresh}(f)$. Let $\Gamma' = \Gamma[f \mapsto \textbf{Fut}\langle B\rangle]$. Obviously $\Gamma' \vdash f = m(e, \overline{e'}, \overline{e''})$ **after** $\overline{f?}; x = f.\textbf{get}$.
– 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** $\overline{f?}$: $\textbf{Fut}\langle B\rangle$ and by T-Assign, $\Gamma(x) = \textbf{Fut}\langle B\rangle$. 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 $\Gamma' = \Gamma[f \mapsto \textbf{Fut}\langle B\rangle]$. Since $\texttt{fresh}(f)$ 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 $\Gamma(f) = \textbf{Fut}\langle B\rangle$ and $\Gamma(\overline{v}) = \overline{T}$. Let $\overline{x}$ be the formal parameters of m in C. Certainly, the auxiliary function $\texttt{bind}(o, f, m, \overline{v}, C)$ returns a process $\{l[\overline{B\ x\ v}, \textbf{Fut}\langle B\rangle]\}$ which is well-typed, and it follows that $\Gamma \vdash_R obj(o, a, p, q \cup \{\texttt{bind}(o, f, m, \overline{v}, C)\})$ **ok**.