Resource Contracts for Active Objects

Charaf Eddine Dridi, Violet Ka I Pun, and Volker Stolz

Western Norway University of Applied Sciences
{Charaf.Eddine.Dridi, Violet.Ka.I.Pun, Volker.Stolz}@hvl.no

Abstract. Workflows coordinate tasks across departments or organisations, where correct execution depends not only on control dependencies but also on the availability of shared resources. This paper presents ReAct, a resource-aware active object language for workflow modelling. In ReAct, method declarations serve as contracts: they specify alternative resource profiles in their signatures, giving methods multiple execution options when resources are limited. Methods can be invoked only once their dependency conditions are satisfied; at activation, a feasible resource profile is then selected and allocated. We encode the language in Maude and show how workflows can be executed, simulated, and verified against their declared dependencies and resource requirements.

Keywords: Active objects · Resource allocation · Contracts · Maude · Workflow Models

1 Introduction

A business process is a collection of structured tasks to fulfil a certain goal within an organisation [1], and a workflow [21] describes the arrangement of tasks and resources within a business process. Workflows can sometimes span across multiple departments or even organisations, with multiple workflows running concurrently. These workflows may share resources while one or more tasks in a workflow local in an organisation are depending on the completion of tasks in the concurrent workflows running in different organisations. Planning this kind of workflows is challenging as it requires specific knowledge from multiple domains to have an overview of how resources are shared across multiple workflows as well as how tasks depend on each other.

Therefore, while there exists tools, e.g., Workflow Management Systems (WMS), that are used to model and automate business processes across heterogeneous tasks while enforcing the intended business logic [8,20], the support for capturing the aforementioned concerns for cross-organisational workflows is limited and inflexible [19].

To address this, we proposed in our earlier work [18] a core language, based on [3], that supports specifying the dependency between tasks (modelled as methods) in method declarations. This allows workflow models to be statically checked, using a type system, to ensure that method invocations respect the declared dependencies.

This paper extends the core language in [18], by allowing resource requirements to be specified in the method declaration. Here, resources are referred to discrete and reusable entities (not consumable or continuous quantities), such as doctors, cars, equipment, required for executing a task. The extended language, called ReAct, enables workflow planners to specify not only one, but multiple alternative sets of resources (each alternative is referred to as a resource profile). Such alternative resource profiles can enhance flexibility of method execution when resources are limited as well as enable decision support in selecting resources based on the metrics that need to be optimised.

With this extension, method declarations in our language can be considered as a form of *contracts* that have to be fulfilled before a method can be invoked: methods can start executing only when all the depending tasks have been completed, and at least one of the required resource profiles is available. Such contracts can potentially facilitate workflow planning by automating the coordination of tasks and resources based on the constraints on execution order and required resources provided by domain experts.

We also provide an *original* implementation of a Maude framework, which gives our language a generic, executable operational semantics well suited to workflow execution. The framework captures the behaviour of workflows modelled in our language, and allows us to check if one or all executions respect the conditions specified in the method declaration.

The rest of the paper is organised as follows: Section 2 presents the syntax and operational semantics of ReAct, extending the core language introduced in [18] with explicit resource declarations and resource-aware rules. Section 3 provides a brief overview of rewriting logic and Maude, and introduces a novel executable specification of the extended semantics in Maude. Section 4 shows the execution and simulation results of an example workflow, illustrating how resources and dependencies are managed and how Maude's search functionality supports verification of behavioural properties. Section 5 discusses related work, and Section 6 concludes the paper with future work.

2 Core Language

In this section, we introduce our core language, ReAct, that allows specifying resource requirements for individual tasks in a workflow model. The language is based on active objects [4], which uses the actor model of concurrency and cooperative scheduling. It is an extension of our earlier work [18] enforcing task dependency in workflows. We first present the syntax with an illustrative example and then discuss the relevant semantic rules.

2.1 Syntax

Fig. 1 shows the abstract syntax of ReAct, which extends the core language defined in [18] by having the notion of global resources R and by allowing specifying

$$
\begin{array}{rcl@{\qquad}rcl}
P & ::= & R\ \overline{CD}\ \{\overline{T\ x}\ ;\ s\} & s & ::= & x = rhs \mid \textsf{skip} \mid \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s \\
R & ::= & \emptyset \mid \{(t, \mathcal{A})\} \cup R & & \mid & \textsf{await}\ f? \mid \textsf{return}\ e \mid s\ ;\ s \\
CD & ::= & \textsf{class}\ C\ \{\overline{T\ x}\ ;\ \overline{M}\} & rhs & ::= & e \mid \textsf{new}\ C \mid f.\textsf{get} \\
M & ::= & Sg\ \{\overline{T\ x}\ ;\ s\} & & \mid & e.m(\overline{e})\ \textsf{after}\ fs \mid e!m(\overline{e})\ \textsf{after}\ fs \\
Sg & ::= & T\ m(\overline{T\ x})\ [DP]\ [RP] & e & ::= & x \mid b \mid fs \mid \textsf{this} \\
T & ::= & B \mid \textsf{Fut}\langle B\rangle & fs & ::= & \texttt{True} \mid fts \mid fs \vee fs \\
B & ::= & C \mid \textsf{Bool} \mid \textsf{Int} \mid \textsf{Unit} \mid \ldots & fts & ::= & f? \mid fts \wedge fts \\
DP & ::= & dp \mid DP \vee DP & dp & ::= & C.m \mid dp \wedge dp \\
RP & ::= & rp \mid RP \vee RP & rp & ::= & (t,\ n,\ \mathcal{A}) \mid rp \wedge rp
\end{array}
$$

Fig. 1. Abstract syntax.

resources required RP to execute a task in a workflow at the level of method declaration.

A ReAct program P consists of a resource pool R, class declarations $\overline{CD}$ and a main method. The resource pool is a multiset of pairs $(t, \mathcal{A})$ representing a resource of type t with uninterpreted attributes $\mathcal{A}$. A class declaration has a name C, fields $\overline{x}$ of types $\overline{T}$ and methods $\overline{M}$. A method M is defined by its signature Sg, specifying its name m and return type T, formal parameters $\overline{x}$ of types $\overline{T}$, the task dependencies DP, and the resource requirement RP of method m, where DP and RP are optional, indicated by square brackets [ ]. A method body consists of local variables $\overline{x}$ of types $\overline{T}$ and a sequence of statements s.

Task dependencies DP specified in the method signature are in disjunctive normal form (DNF) and need to be completed prior to invoking method m. We use $C.m$ to denote a depending method m of class C. We model resource requirements RP similarly, where each rp denoted as $(t,\ n,\ \mathcal{A})$, indicating the quantity n of resources of type t with attributes $\mathcal{A}$, where $n \in \mathbb{N}^+$. The signature of methods that do not depend on any other task and not require any resource is written as $T\, m(\overline{T\, x})$.

Example 1. Let $DP = DP_1 \vee DP_2$, $RP = RP_1 \vee RP_2$, where $DP_1 = C_1.m_1$, $DP_2 = C_2.m_2 \wedge C_2'.m_2'$, $RP_1 = (t_1,\ n_1,\ \mathcal{A}_1)$ and $RP_2 = (t_2,\ n_2,\ \mathcal{A}_2) \wedge (t_2',\ n_2',\ \mathcal{A}_2')$. The signature $T\ m(\overline{T\ x})\ DP\ RP$ specifies the constraints on tasks dependency and the resource requirements of method m. Precisely, the signature states that method m is depending on the completion of method m_1 of class C_1 or method m_2 of class C_2 *and* method m_2' of class C_2'; and indicates that the method requires either n_1 resources of type t_1 with attributes $\mathcal{A}_1$ or n_2 resources of type t_2 with attributes $\mathcal{A}_2$ as well as n_2' resources of type t_2 with $\mathcal{A}_2'$.

Consequently, one can consider the signature of a method as a contract specifying the completion of certain tasks and the requirements of resources that needs to be fulfilled prior to executing the method.

Types are standard, where $\textsf{Fut}\langle B\rangle$ is a future type with values of type B. Statements, as well as the right-hand side of an assignment, are similar to those

```
{ (Intern, A1), (Intern, A2), (SeniorNurse, A3),
  (JuniorNurse, A4), (JuniorNurse, A5), (SeniorResident, A6),
  (LabTechnician, A7)
   ... }

 class Hospital {
   Unit registerPatient()
     req (Intern,1,...) ∨ (JuniorNurse,1,...) { ... }

   Unit startTreatmentPlan()
     dep RadiologyUnit.imagingScan ∧ LaboratoryUnit.bloodTest
     req (Intern,1,...) ∧ (JuniorNurse,2,...) { ... } }

 class CardiologyUnit {
   Unit assessPatient()
     dep Hospital.registerPatient
     req (SeniorResident,1,...) ∧ (SeniorNurse,1,...) { ... } }

 class RadiologyUnit {
   Unit imagingScan()
     dep CardiologyUnit.assessPatient
     req (JuniorResident,1,...) ∨ (SeniorNurse,1,...) { ... } }

 class LaboratoryUnit {
   Unit bloodTest()
     dep CardiologyUnit.assessPatient
     req (LabTechnician,1,...) ∨ (Intern,1,...) { ... } }

{ Hospital h = new Hospital();
  CardiologyUnit cu = new CardiologyUnit();
  RadiologyUnit ru = new RadiologyUnit();
  LaboratoryUnit lab = new LaboratoryUnit();

  Fut<Unit> f1 = h!registerPatient();
  Fut<Unit> f2 = cu!assessPatient() after f1?;
  Fut<Unit> f3 = lab!bloodTest() after f2?;
  Fut<Unit> f4 = ru!imagingScan() after f2?;
  Fut<Unit> f5 = h!startTreatmentPlan() after f3? ∧ f4?;
}
```

Fig. 2. Illustrative example.

in typical active object languages, except for method invocations. Each method invocation is associated to a future f, and we use **await** $f?$ to suspend a process until the associated method returns, i.e., $f?$ is evaluated to *true*, and f.**get** to retrieve the value store in f. Methods can be, either synchronously or asynchronously, invoked only after a (possibly empty) set of methods have completed.

To specify the depending methods, we use an **after** clause containing *fs* that is in DNF. To invoke a method, either at least one conjunction *fts* in *fs* is evaluated to true or *fs* is `True`, i.e., no depending method is specified. Remark that both *DP* and *fs* are required to type check whether methods are invoked conforming to the task dependency specified in the method definition and we refer the interested readers to [18] for the details.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid obj \mid invoc \mid res \mid F \mid cn\ cn & p & ::= & \texttt{idle} \mid \{l \mid s\} \\
obj & ::= & o(a,p,q) & q & ::= & \emptyset \mid \{l \mid s\} \mid q \\
invoc & ::= & invoc(o,f,m,\overline{v}) & val & ::= & v \mid \bot \\
res & ::= & \emptyset \mid \{(t,\mathcal{A})\} \cup res & v & ::= & o \mid f \mid b \mid k \\
F & ::= & fut(f,val) & s & ::= & \textbf{cont}(f) \mid \textbf{suspend} \mid \ldots \\
a,\ l & ::= & \epsilon \mid [\ldots, x \mapsto v, \ldots] & rhs & ::= & e.m(\overline{e}) \mid e!m(\overline{e}) \mid \ldots
\end{array}
$$

Fig. 3. Runtime syntax.

We illustrate the syntax with the simple workflow model presented in Fig. 2, which consist of four classes and a main method. For the clarity of the code, we prepend a keyword **dep** to the task dependency and **req** to the resource requirements in the method declaration.

The resource pool is specified in Lines 1–4. The four classes are defined in Lines 6–27, while the main method is defined in Lines 29–38, which first creates the objects of the four classes. The workflow then starts with registering a patient at the hospital (Line 34), which does not have any task dependency, but requires either an `Intern` *or* a `JuniorNurse`, as reflected in the method signature in Lines 7–8. Note that the specific attributes of the resources are omitted here. After the registration, the patient is assessed by the `CardiologyUnit` (Line 35), which requires one `SeniorResident` *and* one `SeniorNurse` (Lines 15–17). Only after the patient is assessed, two tasks can proceed in parallel: a blood test (Line 36) in the `LaboratoryUnit`, requiring a `LabTechnician` *or* an `Intern`, and an imaging scan (Line 37) in the `RadiologyUnit`, requiring a `JuniorResident` *or* a `SeniorNurse` (Lines 20–22), as shown in the method signatures in (Lines 25–27) and (Lines 20–22), respectively. Finally, once both the blood test and the imaging scan complete, the patient can start the treatment plan at the hospital (Line 38), which requires an `Intern` *and* two `JuniorNurse`, as specified in Lines (10–12).

2.2 Semantics

We present in the following first the syntax of the runtime configuration of *ReAct*, and then proceed to the semantics that manages the task dependency and resource requirements for method invocations in the core language.

Runtime syntax. The runtime syntax is defined in Fig. 3. A runtime configuration cn consists of objects, invocation messages, a global resource pool and futures, denoted as obj, $invoc$, res and F, respectively. Each element in the configuration is separated by space. An empty configuration is written as ϵ. An object comprises an object identifier o, a map a associating object fields to values, a running process p and a pool q of processes waiting to run on the object. A process $\{l \mid s\}$ consists of a map l binding local variables to values and a sequence of statements s, or can also be `idle`.

An invocation message $invoc(o, f, m, \overline{v})$ includes the identity of the callee o, the associated future f, the invoked method m, and its actual parameters $\overline{v}$.

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

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

$$\frac{o' = [\![e]\!]_{a \circ l} \quad \overline{v} = [\![\overline{e}]\!]_{a \circ l} \quad f \text{ fresh}}{o(a, \{l \mid x = e \,!\, m(\overline{e}) \ ; \ s\}, q) \rightarrow o(a, \{l \mid x = f \mid s\}, q) \quad invoc(o', f, m, \overline{v}) \quad fut(f, \bot)} \ (\textsc{Async-Call})$$

$$\frac{o' = [\![e]\!]_{a \circ l} \quad o \neq o' \quad f \text{ fresh}}{o(a, \{l \mid x = e.m(\overline{e}) \ ; \ s\}, q) \rightarrow o(a, \{l \mid f = e!m(\overline{e}) \ ; \ x = f.\textsf{get} \ ; \ s\}, q)} \ (\textsc{Sync-Call})$$

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

Fig. 4. Semantics – part I [18].

A future $fut(f, val)$ consists of its identifier f and a value val which is v if the future is resolved or $\bot$ otherwise. The resource pool res is a multiset of pairs $(t, \mathcal{A})$, each of which indicates the type t and attributes $\mathcal{A}$ of a resource.

The statements are extended with **cont**(f) to return control to the caller process and **suspend** to move a running process into the pool of pending processes, while the right hand side of an assignment is extended with method invocations without the **after** clause.

Operational semantics. A selection of representative semantic rules is presented in Figs. 4 and 5. The remaining rules are standard and can be found in Fig. 8 in the appendix. For clarity, the semantic rules only show the components in a runtime configuration that are affected by the reduction steps. Fig 4 shows the rules defined in [18] that handle method invocations based on task dependency. Rules Async-Call-After and Sync-Call-After handle asynchronous and synchronous method invocations, respectively, where task dependency has to be taken into account, by rewriting the invocation to a conditional statement checking the evaluation of *fs* in the **after** clause. If it returns `True`, the invocation is reduced to one without task dependency; otherwise, a **suspend** statement is prepended to the invocation so that the process will be moved to the pool of pending processes. The evaluation of *fs* has been defined in [18] and can be found in Fig. 9 in the appendix.

While task dependencies in method calls are checked explicitly at the point of invocation, resource requirements are handled implicitly only when activating the invocation at the callee object (see later in Fig. 5). Thus, method calls without task dependency are handled as usual, as shown in rules Async-Call and Sync-call, emitting a message $invoc(o, f, m, \overline{v})$, which in turn is used to

$$\frac{\begin{array}{c}(\textsc{Self-Sync-Call})\\ o = [\![e]\!]_{aol} \quad \overline{v} = [\![\overline{e}]\!]_{aol} \quad f' \text{ fresh} \quad f = l(\mathit{destiny})\\ \{l' \mid s'\} = \mathrm{bind}(o,\, f',\, m,\, \overline{v},\, \mathrm{class}(o)) \quad \mathit{ares} = \mathrm{fpr}(l'(\mathit{rr}), \mathit{res}) \neq \emptyset\end{array}}{\begin{array}{c}o(a, \{l \mid x = e.m(\overline{e})\ ;\ s\}, q) \quad \mathit{res}\\ \rightarrow\ o(a, \{l'[\mathit{ar} \mapsto \mathit{ares}] \mid s'\ ;\ \mathsf{cont}(f)\}, q \cup \{l \mid x = f'.\mathsf{get}\ ;\ s\}) \quad \mathit{res} \backslash \mathit{ares} \quad \mathit{fut}(f', \bot)\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Activate-NoResReq})\\ l(\mathit{rr}) = \emptyset\end{array}}{\begin{array}{c}o(a, \mathtt{idle}, q \cup \{l \mid s\})\\ \rightarrow\ o(a, \{l \mid s\}, q)\end{array}} \qquad \frac{\begin{array}{c}(\textsc{Activate-ResAlloc})\\ l(\mathit{ar}) \neq \bot\end{array}}{\begin{array}{c}o(a, \mathtt{idle}, q \cup \{l \mid s\})\\ \rightarrow\ o(a, \{l \mid s\}, q)\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Activate-ToAlloc})\\ l(\mathit{rr}) \neq \emptyset \quad l(\mathit{ar}) = \bot \quad \mathit{ares} = \mathrm{fpr}(l(\mathit{rr}), \mathit{res}) \neq \emptyset\end{array}}{\begin{array}{c}o(a, \mathtt{idle}, q \cup \{l \mid s\}) \quad \mathit{res}\\ \rightarrow\ o(a, \{l[\mathit{ar} \mapsto \mathit{ares}] \mid s\}, q) \quad \mathit{res} \backslash \mathit{ares}\end{array}}$$

$$\frac{\begin{array}{c}(\textsc{Return})\\ v = [\![e]\!]_{aol} \quad f = l(\mathit{destiny})\end{array}}{o(a, \{l \mid \mathsf{return}\ e\ ;\ s\}, q) \quad \mathit{fut}(f, \bot) \quad \mathit{res}\ \rightarrow\ o(a, \{l \mid s\}, q) \quad \mathit{fut}(f, v) \quad \mathit{res} \cup l(\mathit{ar})} \qquad \frac{\begin{array}{c}(\textsc{Self-Sync-Return})\\ f = l(\mathit{destiny})\end{array}}{o(a, \{l' \mid \mathsf{cont}(f)\}, q \cup \{l \mid s\})\ \mathit{res}\ \rightarrow\ o(a, \{l \mid s\}, q)\ \mathit{res} \cup l(\mathit{ar})}$$

Fig. 5. Semantics – part II. Resource-aware extensions of operational rules.

create a process through a binding process by rule Invoc. For a method m defined as $T\ m(\overline{T\ x})\ DP\ RP\ \{\overline{T\ y}\ ;\ s\}$ in class C, we define:

$$\mathrm{bind}(o,\, f,\, m,\, \overline{v},\, C)\ =\ \{\ \mathit{destiny} \mapsto f,\ \mathit{rr} \mapsto RP,\ \overline{x} \mapsto \overline{v},\ \mathit{ar} \mapsto \bot,\ \overline{y} \mapsto \bot \mid s[o \backslash \mathsf{this}]\ \}$$

that returns a process executing method m by binding its local variable *destiny*, required resource profiles rr and formal parameters $\overline{x}$ to future f, RP and $\overline{v}$, respectively, while the allocated resources ar and local variables $\overline{y}$ remain undefined. Remark that DP is not required in the function, and in the case RP is not specified, i.e., no resource is required, rr is mapped to $\emptyset$. Note also that the local variables *destiny*, rr and ar are reserved keywords.

Fig. 5 presents the rules that we extend from those in [18] to manage resources in ReAct. Resource allocation is handled implicitly through activating a process residing in the pending pool, which is controlled by the three Activate rules. To allocate resources to a process, rule Activate-ToAlloc first checks if the process has been granted any resources and if any of the required resource profile is available in the resource pool. Then, it allocates the resources specified in the available profile, by updating the local variable ar, and removes them from the pool. In other words, resource allocation will only take place when the process first becomes active in the callee object.

The evaluation of the set of required resources is handled by the auxiliary function $\mathrm{fpr}(l'(\mathit{rr}), \mathit{res})$ ("feasible profile") as shown in Fig. 6, which returns the (multi)set of selected resources to be allocated *ares*. Activating processes that

$$\mathrm{fpr}(RP, res) = \begin{cases} \mathrm{fpr}(rp, res) & \text{if } RP = rp \\ AR & \text{if } RP = RP' \vee rp \text{ and } AR = \mathrm{fpr}(rp, res) \neq \emptyset \\ \mathrm{fpr}(RP', res) & \text{if } RP = RP' \vee rp \text{ and } \mathrm{fpr}(rp, res) = \emptyset \end{cases}$$

$$\mathrm{fpr}(rp, res) = \begin{cases} \mathrm{fpr}((t,\ n,\ \mathcal{A}), res) & \text{if } rp = (t,\ n,\ \mathcal{A}) \\ AR \cup \mathrm{fpr}(rp', res \setminus AR) & \text{if } rp = rp' \wedge (t,\ n,\ \mathcal{A}) \\ & \quad \text{and } AR = \mathrm{fpr}((t,\ n,\ \mathcal{A}), res) \neq \emptyset \\ \emptyset & \text{if } rp = rp' \wedge (t,\ n,\ \mathcal{A}) \\ & \quad \text{and } \mathrm{fpr}((t,\ n,\ \mathcal{A}), res) = \emptyset \end{cases}$$

$$\mathrm{fpr}((t,\ n,\ \mathcal{A}), res) = \begin{cases} \{(t, \mathcal{A})\} & \text{if } (t, \mathcal{A}) \in res \text{ and } n = 1 \\ \{(t, \mathcal{A})\} \cup AR & \text{if } (t, \mathcal{A}) \in res \text{ and } n > 1 \\ & \quad \text{and } AR = \mathrm{fpr}((t,\ n{-}1,\ \mathcal{A}), res \setminus \{(t, \mathcal{A})\}) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

Fig. 6. Definition of $\mathrm{fpr}(RP, res)$.

do not require any resource or have already been granted the required resources is carried out by Activate-NoResReq and Activate-ResAlloc.

Resource allocation for synchronous self-calls are handled when the methods are invoked, as shown in rule Self-Sync-Call. To make a synchronous self-call, the rule first checks if any of the resource profiles required by the method is available in the resource pool. Then, it allocates the available required resources to the newly created process and appends statement $\mathbf{cont}(f)$ to the sequence of statements s', which is later used to return control to the caller process. Finally, the allocated resources are removed from the resource pool. Allocated resources will only be returned when the corresponding method returns, as seen in rule Return and Self-Sync-Return.

As we can see from the semantics, methods can only be executed if two conditions hold. Firstly, invoking a method needs to fulfil the task dependency. Therefore, to successfully invoke a method, one of the conjunctions in *fs* must be evaluated to true, i.e., all the futures in a conjunction are resolved. Secondly, an invoked method can only start execution if one of the profiles specified in its resource requirements (if any) is satisfied, given a global resource pool.

3 Formalising Active Objects in Maude

Achieving an executable formal specification along with automatic verification is a complex task. As seen in Section 2, the language encompasses three main aspects. Firstly, we define the syntax of ReAct to identify all the entities of the language that constitute its grammar. Secondly, we specify the runtime configuration to capture how these entities are structured during execution. Lastly, we provide the operational semantics that governs the behaviour and describes how the configuration is evolving.

Correspondingly, we need a formal language and toolset capable of expressing rich structures, composing conditional behaviours, and directly executing their specifications. These considerations led us to choose the Maude language [6], as an implementation of rewriting logic that satisfies our requirements. We choose this language for several reasons: (i) it is expressive enough to encode ReAct's static entities and their relationships; (ii) its equational theory provides the necessary predicate logic for defining equations, computational constraints and feasibility checks; (iii) its executable rewriting logic semantics allow rule application to depend on complex guards such as resource availability or future resolution; and (iv) it includes built-in search and model checking facilities that enable automated verification of properties.

3.1 Maude and Rewriting Logic

The Maude language is a high performance language and tool set based on rewriting logic that supports formal specification, execution, and analysis of systems. Its integration of equational logic with rewrite rules enables concise modelling of system behaviour and rigorous reasoning about system properties [5,6]. Formally, a rewrite logic theory is a tuple $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a membership equation logic theory: Σ is the signature that specifies sorts, subsorts, operators and messages, E a set of (possibly conditional) equations, A a set of equational attributes for operators (e.g., associative, commutative), and R a collection of (possibly conditional) rewrite rules.

The different modules in Maude can be implemented using an object-oriented specification that encompasses objects, messages, classes, and inheritance. An object is represented as $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object name, C is an instance of class, a_i are attribute identifiers, and v_i are their corresponding values for $i = 1 \ldots n$. Concurrent states in object-oriented modules are modelled as multisets of objects and messages, and interactions between objects are governed by rewrite rules:

$$\begin{aligned} &\text{crl } [l] : \langle O_1 : C_1 \mid a_{s_1} \rangle \ \ldots \ \langle O_n : C_n \mid a_{s_n} \rangle \ M_1 \ \ldots \ M_m \\ &\quad \Longrightarrow \langle O_{i_1} : C'_{i_1} \mid a'_{s_{i_1}} \rangle \ \ldots \ \langle O_{i_k} : C'_{i_k} \mid a'_{s_{i_k}} \rangle \ M'_1 \ \ldots \ M'_q \quad \text{if Cond.} \end{aligned}$$

3.2 Execution Semantics

In this section, we show how we employ a Maude-based rewrite theory to define the language and the semantics of ReAct. The executable artefact (code and example) is archived in [9]. Formally, the operational semantics of ReAct is implemented in Maude as:

$$\text{def}_{ReAct} = \{\Sigma_{ReAct}, (E \cup A)_{ReAct}, \ R_{ReAct}\}$$

where Σ_{ReAct} defines the structure and data types for the main entities of *ReAct* (objects, processes, method signatures, invocations, resources, etc.); $(E \cup A)_{ReAct}$

	ReAct	Maude
Syntax	Object	`class OBJECT | id : Oid, fields : Int, proc : ProcessState, suspended : ProcessPool .`
	Method	`class METHOD | sig : Oid, body : Oid .`
	Signature	`class SIGNATURE | ret : Int, name : MethodName, params : ParamList, dp : DP, requires : ResourceProfile .`
	Resource	`class RESOURCE | type : String, attrs : AttrSet, state : ResState, ResCost : Int .`

	Resource Profile	`sort ResourceProfile .`
		`op noneProfile : -> ResourceProfile .`
		`op needs : String Int AttrSet -> ResourceProfile [ctor] .`
		`op _and_ : ResourceProfile ResourceProfile -> ResourceProfile [ctor assoc comm id: noneProfile] .`
		`op _or_ : ResourceProfile ResourceProfile -> ResourceProfile [ctor assoc comm id: noneProfile] .`

	Statement	`sort Statement .`
		`op _= _!_(_) : Object Expr Oid Args -> Statement [ctor] .`
		`op _= _._(_) : Object Expr Oid Args -> Statement [ctor] .`
		`op _= _!_(_)after_ : Object Expr Oid Args Object -> Statement [ctor] .`
		`op _= _._(_)after_ : Object Expr Oid Args Object -> Statement [ctor] .`
		`sort Statement .`
		
		`ops skip eos suspend : -> Statement [ctor] .`
		`op await : Oid -> Statement [ctor] .`

	Process	`sort ProcessState .`
	State	`ops idle : -> ProcessState .` `op { _ | _ } : LocalVarList Statement -> ProcessState [ctor] .`

	Future	`sort FutureState .`
		`class Future | value : ValueOption, state : FutureState .`
		`ops unresolved resolved : -> FutureState [ctor] .`

Semantics	Equations	**`Equations`**`: clauseSatisfied(FS), bind(O,F,...), get(F), feasibleProfile(RP, RS)`
	Messages	**`Messages`**`: op invoc(O,F,M,A) : Oid FutureOid Oid Args -> Msg [ctor] .`
	Semantics rules	**`Rewrite Rules`**`:` `crl [rewrite-rule-name] : State => State' if Equation .`
	Properties	**`Model Checking`**`: search [[n, m]] in ModId : initial-state =>* pattern [such that cond] .`

Table 1. Correspondence between ReAct and Maude

specifies the language's equational theory, specifying computations such as message construction, future binding, and the equational properties of resources; and R_{ReAct} defines the dynamic semantics via a set of rewrite rules that capture operational behaviour. These rules include method invocation, process creation and suspension, and resource allocation and release.

The encoding of ReAct into executable and analysable Maude specifications without loss of information is based on the mappings shown in Table 1. The specifications of the grammar is implemented in an object-oriented module named `REACT-SYNTAX`. The operational semantics of the language is encoded as a set of conditional rewrite rules in another object-oriented module called

`REACT-SEMANTICS`. These modules enable us to perform simulation of the behaviour using rewrite engine of Maude.

Table 1 summarises the main sorts, classes, and operators defining ReAct. The class `METHOD` associates a method body with its `SIGNATURE`. The latter captures task dependencies `dp` and required resource profiles `requires`—both are required for method execution—using the sort `ResourceProfile` and its constructors (`noneProfile, needs, _and_, _or_`) to define alternative resource combinations. Statements comprising method bodies are represented by the sort `Statement`, defined by operators including asynchronous `_= _!_(_)` and synchronous method calls `_= _._(_)`, method invocations with dependencies `after`, and control-flow instructions such as `skip`, `return`, `suspend`, and `await`, etc.

The runtime configuration consists of floating entities i.e., objects, messages, resources, and futures. Each object is an instance of `OBJECT` whose process state is built with the sort `ProcessState` and the operator `{_ | _}`, which binds `destiny`, method parameters, and the resource binding to a `LocalVarList` and pairs them with the statement sequence. Invocation messages `invoc(...)` float in the configuration and when consumed by the Invoc rule, create a process that is placed in the `ProcessPool` to await resources or dependency resolution. Method results are represented by `Future` objects (with `value` and `FutureState` = `resolved`/`unresolved`). The global pool of resources is modelled by a set of objects instances of class `RESOURCE` within `RESOURCE-POOL`.

The operational semantics leverages Maude's rewriting engine: sorts, classes, operators, equations, messages, and (possibly conditional) rewrite rules. Together, they form a rewrite theory whose executions are driven by pattern matching and equational evaluation representing the auxiliary equations. Rules have the shape `crl [name] : State => State' if Condition ..` The left-hand side matches entities in the global configuration; the condition is simplified by the auxiliary equations (e.g., resource feasibility or future resolution); the right-hand side updates object fields, futures, process pools, and the resource pool, possibly emitting or consuming coordination messages. Messages (`invoc(...)`, `releaseRes(...)`, etc.) serve as coordination artefacts: one rule can emit a message to signal a request, and another rule consumes it to advance execution.

As an example, Fig. 7 presents the conditional rewrite rule Activate-Alloc. When activation requires resources (recorded in `rr`) and none have yet been allocated (`ar == noneProfile`), the Activate-Alloc rule computes a feasible profile via `feasibleProfile(...)`, updates the pool, moves the process to execution, and stores the chosen profile in `ar`. The application condition of the rule indicates that the process must have resources requested but not allocated.

Since rules operate within an associative-commutative multiset, the rewrite engine of Maude explores them nondeterministically, so the same initial configuration may evolve along multiple interleaving paths, producing a branching execution tree from any given initial configuration. For example, after an `invoc(...)` message is emitted, Maude may immediately apply the `Invoc` rule to suspend the callee's process or postpone it in favour of other available rewrites. Similarly, `activate-alloc` only fires when its guard conditions are satisfied (i.e., when re-

```
crl [activate-alloc] :
  < RP : RESOURCE-POOL | pool : ResS >
  < O : OBJECT | id : OCid, fields : Fld,
  proc : idle, suspended : { LVL ; (rr :== RP) ; ( ar :== noneProfile )
    | S } ; Q >
=>
  < RP : RESOURCE-POOL | pool : reserveResources(feasibleProfile(RP,
    ResS), ResS) >
  < O : OBJECT | id : OCid, fields : Fld,
  proc : { LVL ; (rr :== RP) ; (ar :== feasibleProfile(RP, ResS)) | S },
  suspended : Q  >
if RP =/= noneProfile.
```

Fig. 7. Rewrite rule ACTIVATE-ALLOC

sources become available), competing nondeterministically with other enabled rules for execution.

Maude's rewrite engine serves as the core executor of ReAct operational semantics. It continuously scans the global configuration—composed of objects, messages, and resource pools—matching the left-hand side patterns of conditional rewrite rules, evaluating guard conditions via the underlying equational theory, and applying the corresponding state transformations. Messages such as `invoc(...)` act as explicit staging points, enabling one rule to emit a message that another rule can subsequently consume. Auxiliary equations (e.g., `bind(...)` and `releaseRes(...)`) handle process creation setup and resource release, while rewrite rules directly update object fields, process states, and resource allocations within an executable framework. This inherent concurrency and interleaving of invocation, suspension, resource-aware activation, and execution is precisely what Maude's built-in search and model checking tools can potentially exhaustively analyse. We hence have tool support to ensure that ReAct correctness properties hold across all possible execution paths.

4 Simulation and analysis

This section presents an executable Maude example, which integrates resource consumption and release with the semantics of method calls, to show correct allocation and termination behaviour. To this end, we revisit the example from Fig. 2 inspired by coordination in a hospital emergency department. The workflow models asynchronous medical tasks across different units, subject to both task dependencies and resource constraints.

4.1 Example

The workflow comprises five asynchronous tasks: `registerPatient`, `assessPatient`, `bloodTest`, `imagingScan`, and `startTreatmentPlan`. The workflow starts with `re-`

`gisterPatient` on the `Hospital` object. Once registration completes, `assessPatient` on `CardiologyUnit` is triggered with explicit `after` clauses. Similarly, only after `assessPatient` completes, both `bloodTest` on `LaboratoryUnit` and `imagingScan` on `RadiologyUnit` are triggered and these two tasks run concurrently. When both tasks `bloodTest` and `imagingScan` complete, `startTreatmentPlan` is initiated at `Hospital`.

Each method invocation is an asynchronous task whose activation is delayed until its `after` dependencies are resolved. In addition, each method declares a required resource profile via `requires`, allowing alternatives through `and`/`or` combinators. These contracts are used to test feasibility at activation time; if sufficient matching resources are available, they are allocated atomically and the process moves from `suspended` to `active`.

All methods draw from a shared `RESOURCE-POOL` containing limited personnel with roles and attributes. Each `RESOURCE` specifies a type (e.g., `"Resident"`, `"Intern"`), and their attributes like `(years(5) ; shift("day"))`. We recall that the pool is modelled as a multiset; the attributes are uninterpreted constants, whose only role is exact matching. We use the following initial resource pool:

```
(Intern, (years(2) ; shift("day"))), (JuniorResident, (years(5) ; shift("day"))),
(SeniorResident, (years(10) ; shift("day"))), (Nurse, (years(5) ; shift("day"))),
(JuniorNurse, (years(5) ; shift("day"))), (JuniorNurse, (years(5) ; shift("day"))),
(SeniorNurse, (years(10) ; shift("day"))), (SeniorNurse, (years(10) ; shift("day"))),
(LabTechnician, (years(5) ; shift("day")))
```

4.2 Execution

Using Maude's rewrite command `rewrite in ACTIVE-OBJ-RESOURCE-TEST : init`, the Maude engine performs iterative applications of the operational semantics rules until no further rewrites are possible. Execution completes almost instantly after 453 rewrites, and the system reaches a stable, terminating state, indicating successful execution and resource handling: every object in the final configuration has reached an idle state and no rule of the operational semantics is enabled. Inspection of the final configuration shows that:

1. **All tasks completed with correct resource discipline:** the `RESOURCE-POOL` in the final state is identical to the one in the initial state. Thus, every allocation obtained during execution has been returned to the pool, leaving it ready for subsequent tasks.
2. **All futures resolved:** every declared method call in our example has finished execution, i.e., all associated futures have been resolved and received a value.
3. **No pending invocations or suspended processes:** There are no *invoc* messages in the configuration, and every object has `proc : idle` with no suspended calls. With no calls in progress, all guards satisfied, and no rules enabled, the system has reached termination.

Next, we discuss how Maude's search mechanism allows us to check all possible executions of our system, as the asynchronous method calls give naturally rise to non-determinism.

4.3 Reachability analysis

We use Maude's `search` to systematically explore the state space of ReAct (specified as a rewrite theory). By selecting the search arrow `=>!` (canonical final states), we can target terminating configurations. Pattern matching over futures, resource states, and objects lets us pose precise queries that reveal liveness issues related to resource consumption, leading to deadlocks, that a single rewrite trace might miss.

All Futures Resolved. To check liveness, we searched for final states where all four futures that we declare in the code (`fregRecord`, `fcardioAssess`, `fimagingScan`, `fbloodTest` and `finitiateTreatment`) were resolved:

```
search in ACTIVE-OBJ-RESOURCE-TEST : init =>!
  < fregRecord         : Future | state : resolved >
  < fcardioAssess      : Future | state : resolved >
  < fimagingScan       : Future | state : resolved >
  < fbloodTest         : Future | state : resolved >
  < finitiateTreatment : Future | state : resolved >
  C:Configuration .
```

This query returned 345 solutions, showing that from the initial configuration all futures resolve in canonical final states (`=>!`) across many interleavings. The number of solutions emerges from intermediate rewriting sequence (e.g., message handling and resource release) that produce syntactically distinct but observationally equivalent configurations with respect to futures and resource availability. Note that this is a program-specific property here, and not necessarily true for ReAct programs in general.

Resource Release. Similarly, we use the canonical state arrow `=>!` to check that specific resources are *available* in all terminating states:

```
search in ACTIVE-OBJ-RESOURCE-TEST : init =>!
  < resourcePool | pool :
      (< r4 : RESOURCE | type : "Junior Resident",
                         state : available > : _) >
  C:Configuration .
```

This query succeeds and shows that the `Junior Resident` (identified by `r4` in the Maude model) is always released at termination. Conversely, searching with `state : consumed` produced no solutions, thus verifying that no resource leaks occur in terminating states.

Global pool restoration. To show that *all* terminating states restore the resource pool to exactly what it was in the initial state, we define an observer `poolOf : Configuration -> ResourceSet` that extracts the pool from any configuration, and a predicate `samePool(C)` that checks `poolOf(C) == poolOf(init)` (equality is checked up to the ACU multiset operator of the pool, so order does not matter). We then ask Maude to explore all *final states* reachable from `init` and return only those that *violate* the invariant. Any solution to the query below would be a counterexample:

```
search in ACTIVE-OBJ-RESOURCE-TEST : init =>!
  C:Configuration
  such that not samePool(C) .
```

This search returned *No solution* meaning Maude found no terminating state with a different pool. Therefore, in every terminating run, all reserved resources are released and the final pool is identical to the initial one.

As a separate sanity check, we also requested at least one final state where the invariant holds *and* nontrivial activity occurred (some future resolved and at least one asynchronous call was issued):

```
search [1] in ACTIVE-OBJ-RESOURCE-TEST : init =>!
  < fm : FUTMON | resolved : RF >
  < counter : COUNTER | count : N >  C:Configuration
  such that samePool(C) and (RF =/= noneF) and (N > 1) .
```

Together, these two searches demonstrate that every terminating execution restores the resource pool exactly to its initial multiset.

5 Related Work

Extensive work has been carried out for workflow modelling and numerous formal approaches have been proposed to analyse workflow behaviour as well as resource management.

UML Activity Diagrams [10] are competitive with WMS notations and provide a powerful notation for modelling control and data flows, they also allow limited communications. However, they lack explicit constructs for modelling inter-workflow message exchanges, making cross-organisational workflows difficult to capture precisely. Both BPEL [17] and BPMN [16] offer rich control-flow constructs for modelling workflows; however, the former lacks inter-workflow messaging, and although the latter supports message exchanges, the dependencies between concurrent workflows are not captured, which prevents reasoning about global workflow consistency and dependencies.

Coloured Petri Nets [13] and YAWL [2] are generally comparable in expressivity to many workflow languages, but control-flow modelling for multiple instances and advanced synchronisation remains limited, and inter-workflow communication is partially supported via hierarchical models in CPN-based approaches. Across all these notations resources remain annotations (lanes/roles or tokens) rather than contracts: there is no built-in notion of a resource profile with feasibility, and disciplined release tied to method activation/return.

BPMN-based resource provisioning strategies by Durán *et al.* [11] focus on dynamic allocation at the business process level. Although they provide executable semantics in Maude, resources are not part of the program syntax: resource requirements are not declared in language artefacts, but parameterise scheduling policies at the process level. Similar to this paper, $\mathcal{R}$PL [3] targets cross-organisational workflows and supports explicit notion of the dependency of task execution order at the level of method invocation. Compared to $\mathcal{R}$PL, ReAct not only models task dependency at the level of method invocation, but also allows specifying at the level of method definition, which makes it possible to verify that the workflow behaviour complies with the required execution order. While resources are explicitly handled, i.e., acquired and released,

in $\mathcal{R}$PL, resources in ReAct are managed implicitly when methods are activated and return according to the requirement specified in the method definition. Resources in Real-Time ABS [14] are capacity provided by deployment components that are consumed explicitly using cost/deadline annotations and recovered when time advances, which are quite from how resources are modelled and handled in ReAct.

Cooperative contracts by Kamburjan *et al.* [15] specify a method's pre/post conditions, frames, and overlap-at-suspension clauses, backed by a denotational trace semantics for an active object language. The focus is on deductive specification and verification of contracts around **await** and **get**, not on resource-aware execution, and the scheduling of the underlying language is unaffected.

Certification of time as a resource through type systems has been addressed by Crary *et al.* [7], where a virtual clock is threaded through types to enforce constant or input-dependent step bounds at compile time. In these approaches, running time is abstracted into a consumable quantity (steps or ticks), and the type checker provides verified bounds on this abstract resource. However, they do so by ignoring runtime dynamics: there are no shared pools, no alternative resource profiles, and no allocate–release discipline responsive to feasibility. Moreover, the work does not address the coordination and communication of concurrent workflows.

fBPMN [12] assigns a first-order logic semantics to BPMN, allowing tasks, events, and gateways to be translated into logical formulas for reasoning about control-flow properties such as consistency and soundness. However, it remains focused on branching and sequencing and does not provide an operational account of communication or synchronisation between concurrent processes. In particular, operational semantics for allocation, suspension, activation, and release of resources lie outside the scope of fBPMN's logic-based formalisation.

In contrast, ReAct is a resource-aware active object language that places resource requirements in method declarations and treats them as interface contracts. The latter are enforced by the operational rules during execution; i.e., at activation, the rules check feasibility, select and consume a concrete set of resources, and at return they release exactly that set. Dependencies are expressed as future-based guards and checked before a call proceeds, enabling correct ordering within the workflow under concurrency. Finally, the semantics and runtime are executable in Maude, its rewriting logic supports systematic search to demonstrate workflow termination, respect of dependencies, and absence of resource leaks across all interleavings.

6 Conclusion

In this paper, we present a resource-aware active object language, ReAct, for workflow modelling. The language extends the core language introduced in [18], which focuses on handling task dependency, by allowing specifying resource requirements in method signatures. By stating these two components at the level of method declaration, the method signatures can be seen as contracts that have to be fulfilled prior to execution; specifically, only when the specified depending tasks have completed and when the required resources are available. ReAct allows specifying alternative resource profiles in the method signature, which consequently increases the flexibility of method execution when resources are limited while keeping call sites free of resource details. Thus, such specification in method declarations can potentially contribute to automated workflow planning, where the specifications are provided by domain experts, and we believe that ReAct fills a gap in workflow modelling.

The operational semantics enforces an allocate-at-activation discipline: a method can only be invoked only when its dependency-guards (when present) hold, and the invoked method can only be activated if a feasible set of available resources exists. The selected resources are allocated atomically on activation and released on return. In addition, we develop an *original*, executable Maude semantics which allows us to analyse the behaviour of workflows modelled in ReAct. Using Maude's search functionality on our example, we demonstrated workflow termination and absence of resource leaks.

Future work. As mentioned, ReAct is an extension of the core language introduced in [18], where a type system is proposed to verify that the behaviour of a workflow respects the task dependency specified in the method signature. Therefore, another possible extension would be to integrate this type system in our presented Maude framework to enrich its functionality. Our interest in a type system is also the reason why we are not yet focusing on more dynamic resource consumption, e.g., through parameters, computation, or transformation: while this is trivial to support in the runtime system, it has implications for static analysis where an upper bound has to be derivable.

We are also planning to introduce a priority-aware scheduler that leverages (i) priority levels declared in method definition (e.g., emergency vs. routine), and (ii) attributes of available resources (e.g., cost, availability, usage time) to decide which process should be activated and which concrete resources to consume. The scheduler can use a scoring function that balances priority, resource attributes, and estimated waiting time in the suspended queue while still giving precedence to methods with higher-priority.

Finally, we plan to investigate in how far Maude's built-in temporal logic model checking can be used to verify other behavioural properties of ReAct workflows, such as verifying that at the start of every recursive or iterative execution, the global resource pool has been restored to its initial configuration, in other words, resources are released between iterations and not held indefinitely.

Acknowledgements. 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).

References

1. van der Aalst, W.M.P.: Exploring the process dimension of workflow management, Computing Science Reports, vol. 97/13. Technische Universiteit Eindhoven (1997)
2. van der Aalst, W.M.P., ter Hofstede, A.H.: YAWL: Yet Another Workflow Language. Information systems **30**(4), 245–275 (2005)
3. Ali, M.R., Lamo, Y., Pun, V.K.I: Cost analysis for a resource sensitive workflow modelling language. Science of Computer Programming **225**, 102896 (2023)
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). `https://doi.org/10.1145/3122848`
5. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude manual (version 3.1). SRI International (2020)

6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Crary, K., Weirich, S.: Resource bound certification. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 184–198 (2000)
8. Dourish, P.: Process descriptions as organisational accounting devices: the dual use of workflow technologies. In: Proc. 2001 ACM Intl. Conf. on Supporting Group Work. pp. 52–60 (2001)
9. Dridi, C.E., Pun, V.K.I, Stolz, V.: Resource contracts for active objects. Source code: https://github.com/selabhvl/maude-active-objects (10 2025). https://doi.org/10.5281/zenodo.17305152
10. Dumas, M., ter Hofstede, A.H.M.: UML activity diagrams as a workflow specification language. In: Gogolla, M., Kobryn, C. (eds.) Proc. 4th Intl. Conf. on «UML» 2001 – The Unified Modeling Language, Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 76–90. Springer (2001). https://doi.org/10.1007/3-540-45441-1_7
11. Durán, F., Rocha, C., Salaün, G.: Resource provisioning strategies for BPMN processes: Specification and analysis using Maude. Journal of Logical and Algebraic Methods in Programming **123**, 100711 (2021)
12. Houhou, S., Baarir, S., Poizat, P., Quéinnec, P., Kahloul, L.: A first-order logic verification framework for communication-parametric and time-aware BPMN collaborations. Information systems **104**, 101765 (2022)
13. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009). https://doi.org/10.1007/B95112
14. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logical and Algebraic Methods in Programming **84**(1), 67–91 (2015)
15. Kamburjan, E., Din, C.C., Hähnle, R., Johnsen, E.B.: Behavioral contracts for cooperative scheduling. In: Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY, LNCS, vol. 12345, pp. 85–121. Springer (2020)
16. Michele, C., Alberto, T.: BPMN: An introduction to the standard. Comput. Stand. Interfaces **34**, 124–134 (2012)
17. Ouyang, C., Dumas, M., ter Hofstede, A.H., van der Aalst, W.M.P.: From BPMN process models to BPEL web services. In: Proceedings of 2006 IEEE International Conference on Web Services (ICWS'06). pp. 285–292. IEEE (2006)
18. Pun, V.K.I, Stolz, V.: Enforced dependencies for active objects. In: Active Object Languages: Current Research Trends, LNCS, vol. 14360, pp. 359–374. Springer (2024)
19. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems – Challenges, Methods, Technologies. Springer (2012). https://doi.org/10.1007/978-3-642-30409-5
20. Senkul, P., Toroslu, I.H.: An architecture for workflow scheduling under resource allocation constraints. Information Systems **30**(5), 399–422 (2005)
21. Workflow Management Coalition: Workflow Management Coalition Terminology & Glossary (1999), https://wfmc.org/wp-content/uploads/2022/09/TC-1011_term_glossary_v3.pdf

A Appendix

In this appendix, we present semantics omitted from Figs. 4 and 5 and the definition of the evaluation $[\![fs]\!]_F$ in Fig. 9.

$$\frac{[\![e]\!]_{a\,o\,l,F} = \texttt{True}}{o(a, \{l \mid \textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2 \ ; s\}, q)\ F \rightarrow o(a, \{l \mid s_1 \ ; s\}, q)\ F} \quad \text{(If-True)}$$

$$\frac{[\![e]\!]_{a\,o\,l,F} = \texttt{False}}{o(a, \{l \mid \textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2 \ ; s\}, q)\ F \rightarrow o(a, \{l \mid s_2 \ ; s\}, q)\ F} \quad \text{(If-False)}$$

$$\frac{v \neq \bot}{o(a, \{l \mid \textsf{await } f? \ ; s\}, q)\ fut(f, v) \rightarrow o(a, \{l \mid s\}, q)\ fut(f, v)} \quad \text{(Await-True)}$$

$$\frac{v = \bot}{o(a, \{l \mid \textsf{await } f? \ ; s\}, q)\ fut(f, v) \rightarrow o(a, \texttt{idle}, q \cup \{l \mid \textsf{await } f? \ ; s\})\ fut(f, v)} \quad \text{(Await-False)}$$

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

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

$$\frac{cn \rightarrow cn'}{cn\ cn'' \rightarrow cn'\ cn''} \quad \text{(Context)}$$

$$o(a, \{l \mid \textsf{skip} \ ; s\}, q) \rightarrow o(a, \{l \mid s\}, q) \quad \text{(Skip)}$$

$$o(a, \{l \mid \textsf{suspend} \ ; s\}, q) \rightarrow o(a, \texttt{idle}, q \cup \{l \mid s\}) \quad \text{(Suspend)}$$

Fig. 8. Semantic rules [18] omitted from Figs. 4 and 5. Assignments for fields and local variables are typical and omitted). The auxiliary function atts(C, o') returns the default values of the fields of class C and o' is the value for **this**.

$$[\![fs]\!]_F = \begin{cases} [\![fs']\!]_F \vee [\![fts]\!]_F & \text{if } fs = fs' \vee fts \\ [\![fts]\!]_F & \text{if } fs = fts \end{cases}$$

$$[\![fts]\!]_F = \begin{cases} [\![fts']\!]_F \wedge [\![f?]\!]_F & \text{if } fts = fts' \wedge f? \\ [\![f?]\!]_F & \text{if } fts = f? \end{cases}$$

$$[\![f?]\!]_F = \begin{cases} \texttt{True} & \text{if } fut(f, v) \in F \wedge v \neq \bot \\ \texttt{False} & \text{otherwise.} \end{cases}$$

Fig. 9. Definition of the evaluation $[\![fs]\!]_F$ [18], with minor adjustment due to updated syntax.