Science of Computer Programming 76 (2011) 37–62

Contents lists available at ScienceDirect

Science of Computer Programming

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

simpA: An agent-oriented approach for programming concurrent applications on top of Java

Alessandro Ricci *, Mirko Viroli, Giulio Piancastelli

DEIS, Alma Mater Studiorum, Università di Bologna, via Venezia 52, 47023 Cesena, Italy

ARTICLE INFO

Article history:
Received 2 April 2008
Received in revised form 24 June 2010
Accepted 27 June 2010
Available online 24 July 2010

Keywords:
Agent-oriented programming
Concurrent programming
Agents and artifacts
Multi-agent systems
Core calculi

ABSTRACT

More and more aspects of concurrency and concurrent programming are becoming part of mainstream programming and software engineering, due to several factors such as the widespread availability of multi-core/parallel architectures and Internet-based systems. This leads to the extension of mainstream object-oriented programming languages and platforms – Java is a main example – with libraries providing fine-grained mechanisms and idioms to support concurrent programming, in particular for building efficient programs. Besides this *fine-grained* support, a main research goal in this context is to devise higher-level, *coarse-grained* abstractions that would help building concurrent programs, as pure object-oriented abstractions help building large component-based programs. To this end, in this paper we present simpA, a Java-based framework that provides programmers with *agent-oriented* abstractions on top of the basic OO layer, as a means to organize and structure concurrent applications. We first describe the application programming interface (API) and annotation framework provided to Java programmers for building simpA applications, and then we discuss the main features of the approach from a software engineering point of view, by showing some programming examples. Finally, we define an operational semantics formalizing the main aspects of this programming model.

1. Introduction

The widespread diffusion and availability of parallel machines given by multi-core architectures is going to have a significant impact in mainstream software development, shedding a new light on *concurrency* and *concurrent programming* in general. Besides multi-core architectures, Internet-based computing and service-oriented architectures/Web services are other main factors that are increasingly introducing concurrency issues in the context of a large class of applications and systems, no longer related only to specific and narrow domains (such as high-performance scientific computing).

As noted in [50], though concurrency has been studied for about 30 years in the context of computer science fields such as programming languages and software engineering, this research has not yet significantly impacted on mainstream software development. As a main example, Java has been one of the first mainstream languages providing first-class native support for multi-threading, with basic low-level concurrency mechanisms. This has been recently extended by means of a new library added to the JDK, including classes that implement well-known and useful higher-level synchronization mechanisms such as barriers, latches and semaphores: this framework provides a *fine-grained* and efficient control on concurrent computations [32]. However, it appears increasingly important to introduce higher-level abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [50].

Accordingly, in this paper we present simpA, a Java-based framework providing programmers with an *agent-oriented* abstraction layer on top of the basic OO layer, to be used as basic building block to define the architecture of concurrent

* Corresponding author. Tel.: +39 0547339217; fax: +39 0547339208.
E-mail addresses: a.ricci@unibo.it (A. Ricci), mirko.viroli@unibo.it (M. Viroli), giulio.piancastelli@unibo.it (G. Piancastelli).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.06.012

applications. Agents and multi-agent systems are considered today a main approach for building complex software systems [26], in particular for engineering intelligent distributed systems. Here we aim at exploiting such an abstraction level for programming concurrent – not necessarily distributed – applications.

A program in simpA can be viewed as a collection of active cooperating entities called *agents*, autonomously fulfilling tasks by interacting in a common *environment*. The environment in simpA is composed of a set of computational entities called *artifacts*, which are organized in *workspaces* and represent resources and tools that agents share and exploit for executing their jobs. Agents interact both by direct communication – based on asynchronous message passing – and by means of the environment, i.e., by exploiting artifacts designed to function as coordination tools (e.g., blackboards, data spaces, semaphores). In particular, artifacts are the core of A&A (Agents and Artifacts), a meta-model grounding simpA that has been recently introduced in the context of agent-oriented programming and software engineering [39].

Analogously to other existing agent platforms such as JADE [7], simpA is realized as a framework on top of the Java language, which is conceptually independent from the OO layer: while agents and artifacts deal with the organization of concurrency abstractions, objects (and the algorithms within) are used to define the computational data structures to be used when programming agents'/artifacts' state and interaction. The main advantage with respect to approaches introducing entirely new languages is that no custom compilers and virtual machines are necessary to develop and run simpA programs: the standard Java tools can be used. Differently from JADE, simpA also relies on the Java Annotation Framework, as do many industrial initiatives such as the server-side framework for Java-based enterprise applications EJB 3.0,[1] the Java application programming interface (API) for XML-Based Web Services JAX-WS 2.0,[2] and the WADE workflow-oriented framework based on JADE [11]. Annotations allow us to conceive Java language extensions by simply tagging classes, fields, and method declarations of standard Java programs, thus enhancing expressiveness and flexibility while retaining the compatibility of standard library approaches.

The main contribution of our proposal with respect to existing "state of the art" approaches to concurrency – in particular with respect to *actors* [1] and active objects [29] – resides in the *level of abstraction* introduced to design and then implement (concurrent) programs, which we root in the concepts of agent, artifact and workspace. Generally speaking, the agent approach promotes a *decentralized mindset* in programming: that is, a way of solving problems and then designing systems based on decentralization of control and of the consequent interaction and coordination of the autonomous parts resulting from decentralization, the benefits and power of this approach being nicely summarized originally by Resnick in his well-known book [42]. Decentralization of control is at the very heart of concurrent programming, and actually is a basic ingredient also of actor-based and process-based approaches (like Erlang [5,28], for instance). Then the issue that we aim at tackling with simpA is: what kind of core abstractions can be devised to adopt and promote a decentralized mindset in computer programming? We believe that such abstractions should (i) hide as much as possible the low-level mechanisms typically found in multi-threaded programming (e.g. semaphores), and then (ii) help in reducing the gap from design to implementation. In this perspective, agents and artifact-based environment abstractions as introduced with simpA both make it possible to abstract completely from low-level issues related to multi-threading and synchronization, and help to bridge the gap between design and implementation so that their features can find a direct coding counterpart.

In simpA, agents can be used to directly implement entities that pro-actively fulfill *tasks*, while artifacts implement those components of the system which – instead of being effectively modeled as task-oriented entities – act as resources that are shared and used to achieve those tasks. Artifacts can span different scales of complexity, from a simple counter, to a complex data repository (e.g., a database). Artifacts can also be used to effectively model and implement tools to support indirect agent communication and coordination, such as blackboards and tuple spaces. This is a main difference with respect to actor-like approaches, which provide a single abstraction (the actor) to model every component of the system, and a single model (asynchronous message passing) for interaction.

In this paper, we present the simpA framework (Section 2), discuss its key aspects and provide examples of application (Section 3), formalize the operational semantics of a subset of simpA (Section 4) with the primary goal of clarifying its most important and subtle aspects, and finally discuss related works (Section 5). This paper develops on the following existing works: in [44] the simpA annotation framework is presented first, in [47] we motivated the agent-oriented abstractions used in simpA, while in [45] an earlier version of the operational semantics is presented. The present paper consolidates such works including a more thorough description of simpA, as well as new and more articulated examples. Also, we include a formalization of a larger subset of the language, tackling crucial aspects of agent and artifact abstract machines.

2. simpA overview

In this section we present an overview of the simpA framework and its programming model. We begin with the presentation of its main concepts (Section 2.1), then agent programming (Section 2.2), artifact programming (Section 2.3), and finally the details of agent–artifact interaction (Section 2.4). The simpA technology is open source:[3] the source code presented in the paper as well as other more complex examples can be found in the simpA distribution.

1 JSR 220, http://jcp.org/en/jsr/detail?id=220.

2 JSR 224, http://jcp.org/en/jsr/detail?id=224.

3 http://simpa.sourceforge.net.

2.1. Main concepts

Agents and artifacts are the basic *coarse*-grained building blocks to structure an application in simpA: a simpA application (system) is viewed as a collection of agents working in one or multiple workspaces, each containing a dynamic set of artifacts, representing resources and tools that agents create, share and use at runtime. Agents and artifacts have a logical name – specified by the programmer at creation time – which must be unique inside the workspace, and have a unique system identifier generated by the simpA platform.

Agent — Agents are autonomous components interacting with their environment. Autonomous here means that they fully encapsulate the control of their behavior, which is directed towards the achievement of some task(s), as described in the agent program. Agents do not have an interface like objects: the interaction with the environment is realized by means of *effectors* (to execute actions) and *sensors* (to collect events from the environment). Agent actions in simpA account for sending messages to other agents[4] and for interacting with artifacts, which are the building blocks composing the environment.

Artifacts — Dually to the agent abstraction, an artifact is a function-oriented and non-autonomous component, designed to provide some functionality that can be exploited by agents, and encapsulating the low-level mechanisms required to avoid interferences during their concurrent use. The functionality of an artifact is structured in terms of *operations*, exposed altogether in the so-called artifact's *usage interface*. An agent executes an operation by acting over the *usage interface*, so as to trigger the execution of a specific operation. The execution of an operation inside the artifact typically results in an update of the artifact's inner state and in the generation of one or multiple *observable events*, i.e., signals carrying some data that can be perceived by the agent that is using the artifact and by the agents that are currently observing the artifact. Besides generating observable events when executing operations, artifacts can exhibit an observable state composed of a dynamic set of properties whose value can be inspected by agents without executing operations. Therefore, artifacts represent controllable and observable devices that agents can use (i) by acting on their usage interface, (ii) by perceiving observable events generated by operation executions and (iii) by observing the value of their observable properties. Also, artifacts can be *linked* together so as to compose connected environments — though this feature is not further discussed in this paper for it is not part of the core of the approach.

Workspaces — Workspaces are logical containers, used to structure the overall environment where the agents play, and representing loci of activities. The environment can be distributed among different network nodes, each node hosting one or multiple workspaces — a single workspace cannot be distributed among multiple nodes. Agents can dynamically join and concurrently work in multiple workspaces, possibly in distinct nodes. By default, a simpA application has a single workspace, called `default` (and this is the main setting we consider in this paper): other workspaces can be created at runtime by means of a dedicated API.

From an implementation viewpoint, a simpA *program* is composed of a set of agent and artifact templates, defining the structure and behavior of specific kinds of agent/artifact, and a set of (Java) classes, representing the basic data types used in agent/artifact definition. Agent and artifact templates are implemented as annotated Java classes, using a one-to-one mapping: an agent template is defined by a single class, and the same holds for artifact templates. The entry point of the program is one boot agent, whose name and template must be specified when launching the program itself either by means of a standalone launcher tool or by means of the simpA API (helper method `launchApplication` in class `alice.simpa.SIMPALauncher`). That method creates a workspace where the application agents and artifacts will be initially contained along with the spawned *boot agent*.

2.2. Programming agents

From an abstract point of view, an agent in simpA can be thought of as an *interpreter* of a program, specified by the agent designer, which contains the description of the *activities* that the agent has to execute in order to fulfill its tasks. The agent program is described in the agent template, which is implemented as a class extending the `alice.simpa.Agent` base class.

2.2.1. Defining agent activities

Activities described in the agent program can be of two basic types:

- *atomic activities*, which are single blocks of instructions (actions), possibly containing actions to interact with the environment;
- *structured activities*, which are a composition of sub-activities.

By default, an agent program must define at least one activity called `main`, either atomic or structured, that is the first activity executed by the interpreter.

Atomic activities are declared in the agent program as methods with the `@ACTIVITY` annotation, with `void` return type and no input parameters. The body of the method specifies the computational behavior of the agent corresponding to the accomplishment of the activity. Fig. 1(top-left) shows the source code of a `HelloAgent`, with a single activity in which the

[4] Inter-agent communication based on asynchronous message passing is not further discussed in this paper, since it is not a central feature of the simpA approach: interested readers can find more information about this aspect in simpA documentation.

```
public class HelloAgent extends Agent {
  @ACTIVITY void main() {
    ArtifactId cons = lookupArtifact("console");
    use(cons, new Op("println", "Hello, world!"));
}}
```

```
public class MyAgent extends Agent {
  @ACTIVITY_WITH_AGENDA({
   @TODO(activity="a"),
   @TODO(activity="b", pre="completed(a)"),
   @TODO(activity="c", pre="completed(a)"),
   @TODO(activity="d", pre="completed(b),
                           completed(c)")
  }) void main() {}

  @ACTIVITY void a() {
    log("completed a");
  }

  @ACTIVITY void b() {
    waitFor(2500);
    log("completed b");
  }

  @ACTIVITY void c() {
    waitFor(1500);
    log("completed c");
   }

  @ACTIVITY void d() {
    log("completed d");
  }
}
```

```
public class MyAgent extends Agent {
  @ACTIVITY_WITH_AGENDA({
   @TODO(activity="a"),
   @TODO(activity="b",
         pre="completed(a)"),
   @TODO(activity="c",
         pre="completed(a)"),
   @TODO(activity="d",
      pre="memo(y(_,_)),memo(z(_))")
  }) void main() {}

  @ACTIVITY void a() {
    memo("x", 1);
  }

  @ACTIVITY void b() {
    int v = getMemo("x").intValue(0);
    memo("y", v+1, v-1);
  }

  @ACTIVITY void c() {
   int v =  getMemo("x").intValue(0);
    memo("z", v*5);
  }

  @ACTIVITY void d() {
    MemoVar y0 = new MemoVar();
    MemoVar y1 = new MemoVar();
    readMemo("y", y0, y1);
    int z = getMemo("z").intValue(0);
    int w = z *(y0.intValue() +
                y1.intValue());
    log("the result is: " + w);
  }
}
```

Fig. 1. (left) An example of agents with a single atomic `main` activity (top) and with a structured activity (bottom). (right) An agent using memos.

agent uses the `console` artifact – which is available in each workspace by default – to print the "Hello, world!" message in standard output. Method local variables (like `cons` in the example) are used to encode data structures representing the local memory related to the specific activity. Agent actions are supported as protected final methods of the `Agent` class (in the example `lookupArtifact` and `use`, described in detail in Section 2.4): the return value of the method represents the *action feedback*, which is a value which depends on the specific action and contains a description of the successful execution of the action; action failure results in an exception.

Structured activities are basically formed by an atomic activity equipped with an *agenda*: the *agenda* specifies the set of sub-activities composing the activity, referred to as *todo* of the agenda. Each todo specifies the name of the sub-activity to execute, and optionally a precondition. When a structured activity is executed, each todo in the agenda is executed as soon as its precondition holds. Therefore, multiple sub-activities can be executed concurrently in the context of the same (super) activity. A structured activity is declared in simpA by a method with an `@ACTIVITY_WITH_AGENDA` annotation, containing todo descriptions as a list of `@TODO` annotations, each specifying the name and precondition. Such preconditions are expressed in a `pre` property as a boolean expression, with and/or/not logical operators (represented by "`,`", "`;`", "`!`" symbols, respectively) over a basic set of predefined predicates. Essentially, such predicates specify conditions over the current state of the activity agenda (whether the state of some sub-activities is "started", "completed", or "aborted") or check the availability of some data in the agent memory (called *memo space*, described in the next subsection).

As a simple example, the agent reported in Fig. 1(bottom-left) has a `main` activity structured around an agenda composed of four todo: `a`, `b`, `c`, and `d`. Activity `a` is meant to be executed as soon as the main activity starts, activities `b` and `c` are executed in parallel as soon as `a` is completed, and finally `d` is executed when both `b` and `c` have completed.

2.2.2. Agent memory

Agent memory is realized as a private associative store called *memo space*, where the agent can dynamically attach, or associatively read, retrieve, and remove chunks of information called *memo*. A memo is a tuple, identified by a label and characterized by an ordered set of arguments, either bound or not to some data object; in the latter case the memo is said to be partially specified. Instance fields in agent classes are not meant to be used: the memo space is the only data structure adopted to represent agent state, shared by the (sub-)activities.

Each agent is provided with internal actions to atomically and associatively access and manipulate the memo space: `memo` is used to create a new memo with a specific label and a variable number of arguments (which can be `null` or bound to an object instance of some class); `getMemo` and `delMemo` to get/remove a memo with the specified label; `readMemo` and `removeMemo` to read/remove a memo matching the specified values, with the possibility to specify both concrete values and variables, as instance of the class `MemoVar`.

```
public class MyAgent extends Agent {
  @ACTIVITY_WITH_AGENDA({
   @TODO(activity="getTaskTodo", persistent=true),
   @TODO(activity="doTask", pre="memo(task_todo)",
                  persistent=true)
  }) void main(){}

  @ACTIVITY void getTaskTodo() {
    // wait for a task todo
    memo("task_todo");
  }

  @ACTIVITY void doTask() {
    removeMemo("task_todo");
    // do task
  }
}
```

```
public class MyAgent extends Agent {
  @ACTIVITY_WITH_AGENDA({
   @TODO(activity="detectStop"),
   @TODO(activity="getTaskTodo",
   persistent_until="memo(stopped)"),
    ...
  }) void main(){}

  @ACTIVITY void detectStop() {
    ...
    memo("stopped");
  }
  ...
}
```

Fig. 2. (left) An example of agent with cyclic behaviors. (right) Using a condition to specify the todo persistence.

To show how the memo space can be used, we extend the previous example by making each activity exploit the memo actions just described (Fig. 1, right). In the example, the agent attaches and retrieves some memos in the memo space, to share data among activities and store the result of its work. In particular, in activity `a` the agent stores a memo `x(1)`; both in `b` and `c` the agent reads the memo labelled with `x`, and uses its content to create respectively a new memo `y(2,0)` and a memo `z(5)`; finally, in activity `d`, the agent uses both the memos `y` and `z` to compute a new value, logged in output.

A memo can also be used to help the coordination of sub-activities, by exploiting the `memo` predicate in the specification of the precondition in the agenda todo: this is used to test the presence of a specific memo in the memo space and also to associatively retrieve its value, through the use of variables (represented by symbols starting with an underscore or an uppercase letter). In the example, while activities `b` and `c` are triggered by the completion of `a`, activity `d` starts as soon as both a memo matching the template `y(_,_)` and a memo matching the template `z(_)` are found. It is worth remarking that precondition evaluation and matching are not meant to alter the content of memos in the memo space. In particular, matching simply checks if there exists a memo whose arguments are equal – using the `equal` method defined in `Object` class – to the value of the ground arguments specified in the template. Then, variables possibly specified in the template are bound to the corresponding arguments of the selected memo.

Finally, note that different activities can change/inspect the memo space concurrently: nevertheless, no race condition can occur in individual memo inspection/update since the memo-related actions are atomic. Races can occur, however, in the case of wrongly designed agendas, including the concurrent execution of multiple sub-activities doing multiple operations on the same memo(s). This happens when the presence of dependencies among sub-activities is not recognized and/or pre-conditions are wrongly specified. Accordingly, we believe that the automated analysis of preconditions and memo space access inside activities can help in finding these kinds of error — which is a subject of our future works.

Concerning deadlocks, note that actions to read/retrieve memos are *not* blocking: they simply fail if no memo is found, so no deadlocks can occur.

2.2.3. Agents with cyclic behavior

Agents are often used to perform repetitive tasks, such as continuously monitoring resources and executing actions as soon as a condition is verified. A direct support for this kind of tasks is provided by *persistent todos*. A todo can be specified to be persistent by setting its boolean `persistent` property to `true` inside the annotation: in that case, once the todo has been executed, it is re-inserted in the agenda so as to be possibly executed again — that is, when its precondition will be positively evaluated again. An example is shown in Fig. 2(left): by means of persistent todos `getTaskTodo` and `doTask` in the `main` activity, the agent repeatedly acquires a new task to be executed and serves it concurrently with the other activities in execution.

For articulated activities, instead of the simple `persistent` property, two properties can be specified, `persistent_if` and `persistent_until`. The former specifies the condition that must hold (when the related activity completes) to consider the todo persistent; dually, the latter specifies the condition until which the todo is considered persistent. An example of usage of this feature is shown in Fig. 2(right). In this case the `getTaskTodo` task is repeated until a memo `stopped` is found, which is meant to be inserted by the `detectStop` activity.

2.3. Programming artifacts

The structure and behavior of an artifact is described in the artifact template, which is implemented as a class extending the `alice.cartago.Artifact` base class.[5]

[5] `cartago` is the name of the separate platform supporting artifact-based environments: simpA extends CArtAgO by providing support for programming and executing the agents living in such environments.

```
public class Counter extends Artifact {

  @OPERATION void init(){
    defineObsProperty("count",0);
  }

  @OPERATION void inc(){
    int count = getObsProperty("count").intValue();
    updateObsProperty("count",count+1);
  }
}
```

```
public class Counter extends Artifact {
  int count;

  @OPERATION void init(){
    count = 0;
  }

  @OPERATION void inc(){
    count++;
    signal("count_incremented",count);
  }
}
```

Fig. 3. (left) A simple *counter* artifact example. (right) An alternative version of the counter without observable properties, generating observable events only.

2.3.1. Artifact structure and behavior

Artifact structure is defined by: (i) a fixed set of variables encoding the non-observable inner state of the artifact, directly implemented by the instance fields of the class; and (ii) a dynamic set of observable properties, which are added/removed and their value inspected/changed by means of a proper set of primitives implemented as protected methods of the `alice.cartago.Artifact` base class.

Artifact behavior is defined by a set of operations, whose execution is triggered by agents through the usage interface. Operations can be *atomic* or *structured*. Atomic operations are characterized by a single block of instructions (called the *operation step*), which is executed atomically as soon as the operation is triggered, and are declared by methods with `void` return type and annotation `@OPERATION`. The name and parameters of such methods declare the name and parameters of the operation that the agent must specify in order to trigger its execution. Structured operations involve the execution of multiple operation steps and – as will be detailed in Section 2.3.3 – are useful in implementing long-term controllable computations, typically useful when coding coordination artifacts. At a given time, only one operation step can be in execution inside the artifact, analogously to entries in monitors; however, multiple operations can be in execution by means of step interleaving. Among the operations, `init` is executed by default when an artifact is instantiated, playing the role of constructors in OO programming.

As a first example, we consider a *counter* artifact, which displays an integer value and provides means to increment it (see Fig. 3, left): to this end, the artifact has a single observable property called `count` displaying the current value of the counter, and a single usage interface control called `inc`, used to increment the counter. An example application for this artifact could be in allowing a team of agents to jointly keep track of the number of hits found when searching for Web pages that contain some information. Beside `inc` and `init` operations, observable properties are defined (typically during artifact initialization) by means of the `defineObsProperty` primitive, specifying the property name and initial value. The observable property `count` is read and updated by the execution of the `inc` operation through available primitives (`getObsProperty`, `updateObsProperty`). In this case, the artifact has no inner state variables.

In this minimal example, no observable events are explicitly generated by artifact operations: however, some observable events are generated by default; in particular, `op_exec_started(`*`OpName`*`)` is generated as soon as the operation started, `op_exec_completed(`*`OpName`*`)` as soon as the operation completed, and an event *`PropName`*`(`*`PropValue`*`)` each time an observable property whose label is *`PropName`* is updated (`count` in this case) with a new value. A variant of the counter without observable properties, purely based on observable events, is shown in Fig. 3(right). The `signal` primitive is supplied for generating events (signals) that can be observed by the agent using the artifact – i.e., the agent that executed the operation – and by all the agents that are observing it. An observable event is characterized by a label describing its kind and possibly an object representing its content. For this counter, an observable event `count_incremented(C)` is explicitly generated in the `inc` operation.

As will be clarified in Section 2.4.2, observable events generated by an artifact are atomically notified to agent *sensors*, which are kinds of private buffers that agents use to manage perceptions. In particular, an observable event is notified to (a) the sensor possibly specified by the agent that executed the operation causing the generation of the event (by means of a `use` action, described in Section 2.4.1), and to (b) the sensors specified by all agents which explicitly manifested their intention to observe the artifact (by means of a `focus` action, Section 2.4.3). So, signals of artifacts which are not observed by any agents, and which are generated by an operation executed by an agent without specifying a sensor, are lost. Conversely, signals collected in sensors are piled up until agents retrieve them by means of a proper internal action (`sense`, Section 2.4.2).

2.3.2. Usage interface controls with guards

For each usage interface control, a *guard* can be specified: that is, a condition that must be true in order to actually enable the control. By default, the control is enabled: a guard then can be useful to constrain the availability of the control to a dynamic condition related to the internal state of the artifact. If a control is enabled, then it can be successfully triggered by a use action of an agent; otherwise, the use action is suspended (more details on this can be found in Section 2.4). Guards are implemented as boolean methods with the `@GUARD` annotation and are associated to a usage interface control by specifying the attribute `guard` in the `@OPERATION` annotation triggered by the control.

```
public class BoundedBuffer<T> extends Artifact {
  LinkedList<T> items;
  int nmax;

  @OPERATION void init(int nmax) {
    items = new LinkedList<T>();
    this.nmax = nmax;
  }

  @OPERATION(guard="notFull") void put(T obj) {
    items.add(obj);
    signal("item_inserted");
  }

  @OPERATION(guard="notEmpty") void get() {
    T item = items.removeFirst();
    signal("new_item_available", item);
  }

  @GUARD boolean notEmpty() {
    return items.size() > 0;
  }

  @GUARD boolean notFull(T obj) {
    return items.size() < nmax;
  }
}
```

```
public class Barrier extends Artifact {
  int nmax;
  int count;

  @OPERATION void init(int nmax) {
    this.nmax = nmax;
    count = 0;
  }

  @OPERATION void await() {
    count++;
    nextStep("awaitFinished");
  }

  @OPSTEP(guard="barrierPointAchieved")
  void awaitFinished() {
    signal("barrier_point_achieved");
    count = 0;
  }

  @GUARD boolean barrierPointAchieved() {
    return count >= nmax;
  }
}
```

Fig. 4. A bounded buffer (left) and a barrier (right) implemented as artifacts.

As a simple example, Fig. 4(left) shows the implementation of a bounded buffer artifact, to be used in producer–consumer architectures: the `put` operation control is enabled only if the buffer is not full, and dually the `get` operation control is enabled only if the buffer is not empty.

A guard is called passing the same parameters of the related guarded usage interface control or operation step. This makes it possible to specify guard conditions including also the value of actual parameters, besides the state of the artifact.

2.3.3. Structured operations

In the previous examples, artifact operations were *atomic*, namely, composed of a single step. However, since artifacts can execute only one step at a time, this would result in a rather inflexible programming environment for concurrency abstractions. Structured operations can hence be implemented as a sequence of operation steps, allowing the concurrent execution of long multiple-structured operations by interleaving the guarded execution of these steps.

Operation steps are implemented by methods annotated with `@OPSTEP`, and can be triggered (enabled) by means of the `nextStep` primitive, specifying the name of the step and a (possibly empty) list of parameters. When no step is executing, a triggered step whose guard evaluates to true is selected and executed — the other steps triggered in the context of the same operation are dropped and never executed.

As a simple example, consider the `Barrier` artifact shown in Fig. 4(right). The operation `await` is composed of two steps: the first is triggered with the invocation of the operation, while the second, named `awaitFinished`, is triggered by the first through the `nextStep` primitive. Once triggered, `awaitFinished` is executed as soon as its `barrierPointAchieved` guard is evaluated to true, that is, only when the artifact variable `count` reaches a value greater or equal to the `nmax` value, specified as a parameter in artifact instantiation. Each time `await` is triggered, `count` is incremented: so the condition will not hold until `await` is triggered `nmax` times. At that point, `awaitFinished` is executed and `await` can then complete, generating the `barrier_point_achieved` observable event.

2.4. Agent–artifact interaction model

2.4.1. Using artifacts

Agents can interact with an artifact by means of the `use` action (see Fig. 5), specifying the selected usage interface control and related required parameters. If the use action succeeds (the control's guard is positively evaluated), then a new instance of the operation linked to the operation control starts its execution inside the artifact. The execution of the operation eventually generates a stream of observable events that may be perceived both by the agent that performed the `use` action and by all the agents that are observing the artifact — as described below.

A concrete example of use is shown in Fig. 6(left): a `CountUser` agent first creates a `Counter` artifact (whose template has been described in Fig. 3), then uses the artifact by executing twice the `inc` operation by means of the `use` action, and finally it waits to perceive the `count_value` event generated by the artifact by means of the `sense` action, discussed in the next subsection.

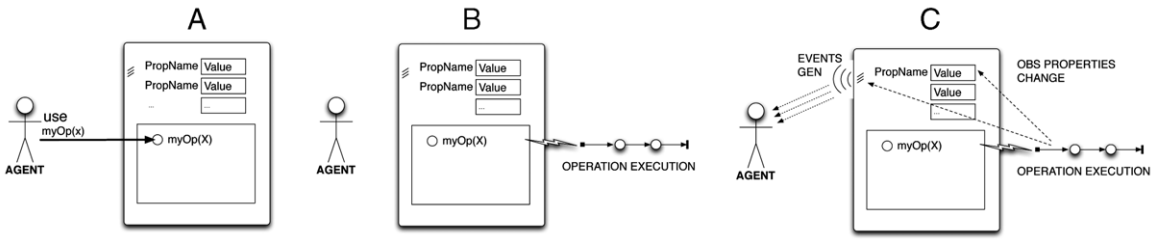

Fig. 5. An agent using an artifact, first acting on the usage interface and then perceiving observable events.

```
public class CountUser extends Agent {

  @ACTIVITY void main() {

    ArtifactId countId =
           makeArtifact("myCount","Counter");

    use(countId, new Op("inc"));

    SensorId s = linkDefaultSensor();
    use(countId, new Op("inc"), s);

    try {
      Perception p =
             sense(s, "count_incremented", 1000);
      long value = (Long) p.getContent();

      ArtifactId cons = lookupArtifact("console");
      use(cons,println("Count value: "+value);

    } catch (NoPerceptionException e) {
      log("No perceptions within a second.");
    }
  }
}
```

```
public class Observer extends Agent {
  @ACTIVITY_WITH_AGENDA({
   @TODO(activity="setup"),
   @TODO(activity="observe",
     pre="completed(setup)",
    persistent_until="memo(ncounts(5))")
   @TODO(activity="finalise",
     pre="memo(ncounts(5))")
  }) void main(){}

  @ACTIVITY void setup() {
    memo(new Memo("ncounts",0));
    ArtifactId c =
           lookupArtifact("myCount");
    SensorId s = getSensor("s0");
    focus(c, s);
  }

  @ACTIVITY void observe() {
    SensorId s = getSensor("s0");
    sense(s, "count_incremented");
    log("new count observed");
    Memo n = delMemo("ncounts");
    int newVal = n.intValue(0)+1;
    memo(new Memo("ncounts",newVal));
  }

  @ACTIVITY void finalise() {
    ArtifactId c =
           lookupArtifact("myCount");
    stopFocussing(c);
  }
}
```

Fig. 6. (left) An agent interacting with an `Counter` artifact by means of `use` and `sense` action. (right) An agent observing the same artifact by means of `focus` action.

The `use` action is executed specifying the identifier of the target artifact, the usage interface control to select (i.e., the operation to be triggered) and its list of parameters. Optionally, one can specify also the identifier of the *sensor* used to collect observable events (details in next subsection), and a timeout. In the example, the first occurrence of `use` triggers the execution of the `inc` operation, without specifying any sensor to collect observable events generated by the operation execution. The second occurrence triggers `inc` again by this time specifying a sensor (identifier).

The execution of a use action involves a *synchronous* interaction between the agent and the artifact: action success means that the operation assigned to the control has started its execution. This contrasts with standard agent–agent communication, which is based, instead, on fully asynchronous communication. If the usage interface control is part of the usage interface but currently is not available (enabled), the `use` action is suspended for a maximum time equal to the specified timeout, then it fails. So a `use` action is suspended when either the artifact is executing some other operation (step) – in that case, all usage interface controls are temporarily disabled – or the guard of an operation control is not satisfied. Then, the execution of operations is completely asynchronous with respect to agent activities: the `use` action does not involve any transfer of control as happens in the case of remote procedure call or method invocation in procedure-based or object-oriented systems.

The `makeArtifact` and `lookupArtifact` actions that appear in the example are provided respectively to instantiate artifacts – specifying the template and the set of parameters needed for its instantiation (which must be compatible with the parameters specified in the `init` artifact operation) – and retrieve the identifier of an existing artifact in the workspace, given its name. Actually, artifact instantiation and discovery are realized by means of artifact use, through a *factory* artifact and a *registry* artifact, available in each workspace: so the actions `makeArtifact` and `lookupArtifact` are not primitive actions but auxiliary actions hiding the interaction based on `use` and `sense` actions with those artifacts.

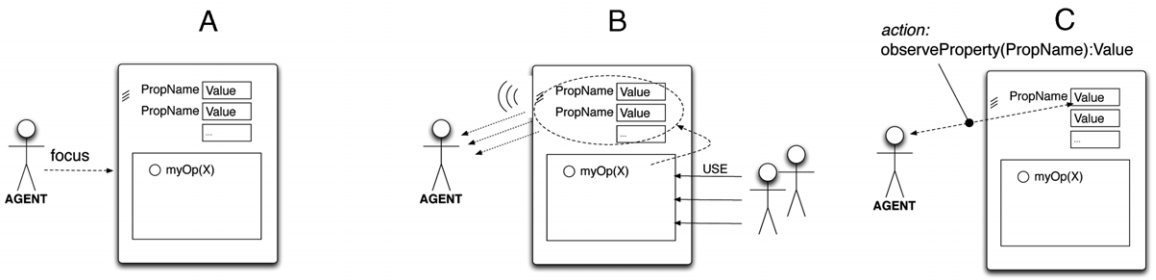

Fig. 7. An agent observing an artifact, by means of a `focus` action and `observeProperty` action.

2.4.2. Perceiving observable events

As seen in previous example, to perceive observable events the agent executing a `use` action can specify a *sensor*, where an artifact's observable events will be collected as soon as they are generated by the artifact. Then, a `sense` action is provided to fetch events from the sensor as soon as needed, possibly specifying filters that match one or more events in the sensor's queue (for data-driven perception) and a timeout.

In the example in Fig. 6(left), the agent waits to perceive a `count_incremented` event, with a time-out of 1000 ms. If `sense` is executed successfully, the event is removed from the sensor and a perception related to that event – represented by an object of class `Perception` – is returned. If no perception is sensed for the duration of time specified, a `NoPerceptionException` is thrown.

Sensors play the role of *perceptual memory* that is explicitly manageable by the agent: agents can use sensors to organize in a flexible way the processing of events, which can be possibly generated by multiple different artifacts and that an agent can be using for different, even concurrent activities. An agent can link (and unlink) any number of sensors, according to the strategy chosen for perceiving events, by means of specific actions: `linkSensor`, `unlinkSensor`, and `linkDefaultSensor`. The latter operation links a new sensor of the default type, which provides a FIFO policy to manage collected events. Operation `getSensor` retrieves the identifier of a sensor given its name (and possibly creates a new one, if no sensor with such a name exists).

2.4.3. Observation

Besides use, *observation* is the other sort of agent–artifact interaction, which accounts for perceiving observable properties and events of an artifact *without using it*, i.e., without acting on its usage interface as in the previous case. To this end, two basic actions are provided, `observeProperty` and `focus`.

The `observeProperty` action simply makes it possible to read the current value of a specific observable property, which is returned directly as feedback of the action. The `focus` action is more complex (see Fig. 7): by executing a focus on a specific artifact (specifying a sensor like in the `use` case), an agent is notified of all the observable events that the artifact will generate from that moment, even if it was another agent that actually caused them — much like event listening in Java. All the observable events generated by the artifact are detected by the sensor and fetched by the agent through a `sense` action whenever it decides to do so. Action `stopFocus` is provided to stop the observation. An example of an agent using `focus` and `stopFocus` is shown in Fig. 6(right): the `Observer` agent continuously observes a `Counter` artifact and prints a message on the console each time a `count_incremented` event is perceived, for five times before stopping the observations.

It is worth noting that the `focus` and `observeProperty` actions do not have any effect on the artifact's state and behavior, the only changes and effects are on the agent side. By contrast, in purely agent-based and actor-based scenarios, for an agent to observe an information belonging to another agent, an explicit message-based request–response interaction protocol between the two agents must be used, with both agents necessarily executing some actions for carrying on the protocol.

3. Programming examples and discussion

The main objective of simpA is to simplify the design and development of concurrent programs: in this section we focus on the main features of the approach related to this aspect.

3.1. Abstraction and separation of concerns

The level of abstraction underlying the simpA approach is meant to ease the design of the application and then to reduce the gap between this design and the implementation. At the design level, by adopting a task-oriented approach as typically promoted, for example, by agent-oriented methodologies [24], the task-oriented and function-oriented parts of the system are identified, leading to the definition of the agents and artifacts as represented by the A&A model, and then to the implementation in simpA. Tasks are units of work to pro-actively achieve or maintain some kind of goal, by executing

some set of actions, possibly affecting different parts of a system. By adopting simpA, tasks are assigned to agents, which have then the responsibility of their execution and control; the description of tasks drives the definition of agent activities and related agenda. Functions can be seen as units of functionality that can be exploited as basic actions to perform tasks. In simpA, functions are directly encapsulated into artifacts, which represent the means – resources and tools – that agents share and use to perform the tasks.

It is worth remarking here on the conceptual distinction between agents and artifacts. On the one hand, agents are *autonomous* entities; that is, they fully encapsulate a state, a computational behavior (structured in activities) and the *control* over it, lacking any interface to govern such activities from the outside. On the other hand, artifacts – as in the case of objects in purely OO approaches – encapsulate a state and a computational behavior, but not the control over this behavior, which is governed instead from the outside through the artifact's exposed usage interface. Inside agents, activities are scheduled and executed according to their own internal agenda(s), specified by the agent programmer. Inside artifacts, operations are triggered and executed as a result of agents' use actions. We believe that this distinction improves the *separation of concerns* in developing (concurrent) decentralized programs: in our case a separation of concerns into agents and the environment (composed of artifacts), the former encapsulating control and the latter being controlled by the former.

3.2. Modularity

The simpA approach aims at providing general means to modularize the behavior of agents (as controlling entities) and artifacts (as controlled entities).

3.2.1. Modularity in agents

The hierarchical model of activities introduced in simpA is meant to mirror the decomposition of a task to execute in subparts, typically conducted at the design stage, with the notion of agenda and todo used to specify in a declarative way the glue among sub-activities. We argue that this is crucial when non-trivial active behaviors are concerned. In simpA, the hierarchical model of activities on the one hand, and the declarative model to specify their dependencies on the other hand, provide a discipline in that direction, improving the level of modularity and encapsulation of active behaviors, which is quite important as soon as agents with articulated tasks are considered.

3.2.2. Modularity in artifacts

Concerning artifacts, the model adopted in simpA makes it possible to specify complex functionalities (operations) without the need to explicitly use lower-level Java mechanisms such as synchronized blocks or wait/notify synchronization primitives. This simplifies – among the others – the implementation of coordination artifacts, which are artifacts providing coordination functionalities to agents. A simple example is the barrier, described in Section 2.3.3. A more complex example is a *tuple space* [18], shown in Fig. 8(left), used by simple master agents and worker agents shown in Fig. 8(right): the former allocates task to be executed (represented by `task_todo` tuples) and retrieves the results (represented by `task_result` tuples); the latter gets the tasks to be executed and provides the results.

On the one hand, mutual exclusion in accessing and modifying an artifact's inner state is guaranteed by having only one operation step executed at a time — non-blocking, atomic accesses to the artifact state are to be encapsulated in a single step. On the other hand, dependencies between operations can be explicitly taken into account and expressed declaratively by defining operation (step) guards. In the tuple space example, for instance, the basic synchronization mechanism provided by blocking operations such as `in` is simply realized by an operation composed of two steps: the second one – which completes the operation execution by removing the tuple – is executed only when a tuple matching the template is found in the tuple set (as specified by the `foundMatch` guard).

It is worth remarking here that the orthogonality with the OO model makes it possible to integrate the agent (artifact) layer and the OO layer in a quite clean way: in the example, the data structures `TupleSet`, `Tuple` and `TupleTemplate` used to implement the `SimpleTupleSpace` artifact do not need any concurrency mechanism, but are rather meant to provide efficient implementation of purely algorithmic data structures involved in tuple management.

3.3. Decentralization and coordination

Besides the individual component level, the approach has been conceived to simplify the development of systems composed of multiple agents that work together, coordinating their activities by exploiting suitable set of artifacts. The features of the programming model – in particular of the agent and environment (artifacts) abstraction – are meant to promote the design and development of decentralized solutions to problems, from both conceptual and pragmatical points of view.

As an example, here we consider the basic well-known *dining philosopher* problem. The problem is about a set of N philosophers (typically 5) sharing N forks for eating spaghetti, sitting at a round table — hence, each philosopher shares his/her left and right forks with one friend philosopher on the left and one on the right. The goal of each philosopher is to live a joyful life, interleaving thinking activity, for which they actually do not need any resources, with eating activity, for which they need to take and use both the forks. The goal of the overall philosophers group is to share the forks fruitfully, and coordinate the access to such resources so as to avoid forms of deadlock or starvation of individual philosophers — e.g.,

```
public class SimpleTupleSpace
                            extends Artifact {
  TupleSet tset;

  @OPERATION void init(){
    tset = new TupleSet();
  }

  @OPERATION void out(Tuple t){
    tset.add(t);
  }

  @OPERATION void in(TupleTemplate tt){
    Tuple t = tset.removeMatching(tt);
    if (t!=null){
      signal("tuple",t);
    } else {
      nextStep("inDone",tt);
    }
  }

  @OPSTEP(guard="foundMatch")
  void inDone(TupleTemplate tt){
    Tuple t = tset.removeMatching(tt);
    signal("tuple",t);
  }

  @OPERATION void rd(TupleTemplate tt){
    Tuple t = tset.removeMatching(tt);
    if (t!=null){
      signal("tuple",t);
    } else {
      nextStep("rdDone",tt);
    }
  }

  @OPSTEP(guard="foundMatch")
  void rdDone(TupleTemplate tt){
    Tuple t = tset.readMatching(tt);
    signal("tuple",t);
  }

  @GUARD boolean foundMatch(TupleTemplate tt){
    return tset.hasTupleMatching(tt);
  }
}
```

```
public class Master extends Agent {

  @ACTIVITY_WITH_AGENDA({
      @TODO(activity="assigningTask",
            persistent=true),
      @TODO(activity="gettingResult",
            persistent=true),
  }) void main(){}

  @ACTIVITY void assigningTask() {
    ArtifactId bag = lookupArtifact("bag");
    SensorId sid = getSensor("s0");
    double value = generateTask();
    Tuple todo = new Tuple("task_todo",
                           value);
    use(bag, new Op("out", todo));
    TupleTemplate result =
            new TupleTemplate("task_result",
                              value,null);
    use(bag, new Op("in", result),sid);
  }

  @ACTIVITY void gettingResult() {
    SensorId sid = getSensor("s0");
    Perception p = sense(sid,"tuple");
    Tuple t = (Tuple)p.getContent(0);
    log("got result: "+t.doubleContent(0)+
          " ==> "+t.doubleContent(1));
  }

  private double generateTask(){...}
}

public class Worker extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="consuming",
          persistent=true)
  }) void main(){}

  @ACTIVITY void consuming() {
    SensorId sid = getSensor("s0");
    ArtifactId bag = lookupArtifact("bag");
    TupleTemplate todo =
            new TupleTemplate("task_todo",
                    (Object)null));
    use(bag, new Op("in", todo, sid);
    Perception p = sense(sid,"tuple");

    Tuple t = (Tuple)p.getContent(0);
    double value = t.doubleContent(0);
    double result = Math.sqrt(value);
    Tuple res = new Tuple("task_result",
                          value,result);
    use(bag, new Op("out", res));
  }
}
```

Fig. 8. (left) Implementation of a simple tuple space as an artifact. `Tuple`, `TupleTemplate` and `TupleSet` – not reported – are flat data types, with no synchronization mechanisms, implementing tuples, tuple templates (providing a matching operation on tuples) and the collection managing the tuples. (right) Implementation of simple master and worker agents in a bag-of-task architecture, interacting by means of the tuple space.

when all philosophers have one fork each. The social constraint of the group is that a fork cannot be used simultaneously by more than one philosopher.

A natural way to devise a solution using simpA is to use agents to implement the philosophers and suitable designed artifacts to model the resources used by the philosophers. A first solution – similar to solutions based on monitors – accounts for using a single *table* artifact functioning as resource (fork) manager and coordination medium, encapsulating the coordination policy that handles mutual exclusion for the access of individual forks and that avoids deadlock situations. Clearly such a solution suffers from a centralization problem, due to the single table artifact.

A fully decentralized solution can be easily conceived by replacing the table with a proper set of independent artifacts, each encapsulating a piece of functionality related to resource access and coordination, and exploiting the basic properties of artifacts use and observation.[6]

[6] Actually, different kinds of decentralized solutions can be devised for the dining philosopher problem: the one adopted here is not the simplest, but one of the most effective in showing how to exploit basic agent and artifact features for realizing decentralized coordination.

```
public class Fork extends Artifact {
  private boolean isAvail;
  @OPERATION void init(){
    isAvail = true;
  }
  @OPERATION(guard="isForkAvailable")
  void acquire(){
    isAvail = false;
  }
  @GUARD boolean isForkAvailable(){
    return isAvail;
  }
  @OPERATION void release(){
    isAvail = true;
  }
}

public class TurnDisplay extends Artifact {
  @OPERATION void init(int startValue){
    defineObsProperty("turn",startValue);
  }
  @OPERATION void next(){
    int t = getObsProperty("turn").intValue();
    updateObsProperty("turn",t+1);
  }
}

public class TicketDispenser extends Artifact {
  int ticket;
  @OPERATION void init(){
    ticket = 0;
  }
  @OPERATION void getTicket(){
    signal("ticket",ticket++);
  }
}
```

```
public class Waiter extends Agent {

  @ACTIVITY void main() throws Exception {
    int nphilo = 5;
    int nforks = 5;

    for (int i = 0; i < nforks; i++){
      makeArtifact("fork-"+i,"Fork");
    }

    makeArtifact("ticket_dispenser",
                 "TicketDispenser");
    makeArtifact("turn_display",
          "TurnDisplay",
          new ArtifactConfig(nphilo-1));

    for (int i = 0; i < nphilo; i++){
      String lfork = "fork-"+i;
      int rid = (i+1) % nforks;
      String rfork = "fork-"+rid;
      String agentName = "philo-"+i;
    spawnAgent(agentName, "Philosopher",
        new AgentConfig(lfork,rfork) ));
    }

  }
}
```

Fig. 9. Source code of the artifacts (left) and waiter agent (right) in the decentralized solution to the dining philosopher problem.

In particular, a `Fork` artifact can be introduced to wrap the access to each fork (resource); a `TicketDispenser` artifact can be introduced to rule each philosophers' access to the table, by distributing tickets; and finally a `TurnDisplay` artifact can be used for keeping track of (and make it observable) which ticket number can be served. Each philosopher must first get a ticket from the `TicketDispenser`; then, after retrieving a ticket, a philosopher can get the forks and eat as soon as the turn reported by the `TurnDisplay` is greater or equal to the ticket number owned by the philosopher.

The source code of the artifacts is reported in Fig. 9(left) — the booting agent is shown on the right. Artifact `Fork` simply provides two operations to acquire and release the resource, `TicketDispenser` to get a new ticket, which is simply a monotonic increasing integer. `TurnDisplay` has an observable property (`turn`) reporting the maximum ticket number that can be served and an operation `next` to advance such a value. The `turn` property is properly initialized in the operation `init` with the value specified by the agent creating the artifact (the `WaiterAgent`), which corresponds to the total number of philosophers minus one. This means that the system allows no more than $N - 1$ philosophers to eat concurrently, which is the condition that makes it possible to avoid deadlocks. Also, the absence of starvation for a philosopher that wants to eat is guaranteed by the use of the ticket dispenser.[7]

The agent program for the philosophers is shown in Fig. 10. Though it appears somehow articulated, it is well modularized: after executing a `setup` activity, the `work` activity accounts for repeatedly fulfilling the `eating` task when the agent has a `hungry` memo, or `thinking` task otherwise. The `eating` activity is further composed of sub-activities, part of them executed in parallel as specified in the agenda. First the agent gets a ticket from the `TicketDispenser`, then it waits for its turn, by observing the `TurnDisplay` waiting to perceive a turn value greater or equal to the agent's current ticket value. When this occurs, the philosopher can finally interact with the two distinct `Fork` artifacts to acquire the resources, by means of two sub-activities, `getLeftFork` and `getRightFork` which can be logically executed in parallel. As soon as both such sub-activities successfully completed (which means that both the forks have been acquired), then the philosopher can eat. After that, the agent can release the resources and cease the turn (by interacting with `TurnDisplay`): also these activities are independent from each other and can be done in parallel.

3.4. Decoupling logical and physical concurrency

By introducing a further level of abstraction over the OO layer and the basic multi-threading mechanisms provided by Java, a simpA program is less efficient than programs exploiting directly fine-grained mechanisms, as those provided by Java

[7] Here we do not consider the case of starvation due to philosophers' misbehavior or failure.

```
public class Philosopher extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="setup"),
    @TODO(activity="work",
          pre="completed(setup)")
  }) void  main() {}

  @ACTIVITY void  setup() {
    Memo memo = getMemo("boot_args");
    String lfork = memo.stringValue(0);
    String rfork = memo.stringValue(1);
    ArtifactId left = lookupArtifact(lfork);
    ArtifactId right = lookupArtifact(rfork);
    memo("forks", left, right);
    ArtifactId disp =
            lookupArtifact("turn_display");
    focus(disp,getSensor("s0"));
    memo("hungry");
  }

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="eating",
         pre="memo(hungry)", persistent=true),
    @TODO(activity="thinking",
         pre="!memo(hungry)", persistent=true)
  }) void  work(){}

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="getTicket"),
    @TODO(activity="waitForMyTurn",
          pre="memo(ticket(_))"),
    @TODO(activity="getLeftFork",
          pre="completed(waitForMyTurn)"),
    @TODO(activity="getRightFork",
          pre="completed(waitForMyTurn)"),
    @TODO(activity="eat",
          pre="completed(getLeftFork),
               completed(getRightFork)"),
    @TODO(activity="releaseLeftFork",
          pre="completed(eat)"),
    @TODO(activity="releaseRightFork",
          pre="completed(eat)"),
    @TODO(activity="ceaseTurn",
          pre="completed(eat)")
  }) void eating() throws Exception {}

  ...
```

```
  ...
  @ACTIVITY void getTicket(){
    SensorId sid = getSensor("s0");
    use(tid, new Op("getTicket"),sid);
    Perception ticket = sense(sid,"ticket");
    memo("ticket",ticket.intContent(0));
  }
  @ACTIVITY void getLeftFork(){
    Memo forks = getMemo("forks");
    ArtifactId left = forks.asArtId(0);
    use(left, new Op("acquire"));
  }
  @ACTIVITY void getRightFork(){ ... }

  @ACTIVITY void waitForMyTurn(){
    Memo ticket = getMemo("ticket");
    int myticket = ticket.intValue(0);
    sense(getSensor("s0"),
            new MyTurnFilter(myticket));
  }
  @ACTIVITY void eat() {
    // eat eat
    delMemo("hungry");
  }
  @ACTIVITY void releaseLeftFork() {
    Memo forks = getMemo("forks");
    ArtifactId left = forks.asArtId(0);
    use(left, new Op("release"));
  }
  @ACTIVITY void releaseRightFork() { ... }

  @ACTIVITY void ceaseTurn() {
    delMemo("ticket");
    ArtifactId tid =
            lookupArtifact("turn_display");
    use(tid, new Op("next"));
  }
 @ACTIVITY void thinking() {
    // mumble mumble
    memo("hungry");
  }
}

public  class MyTurnFilter
            implements IPerceptionFilter {
  private int turn;
  public MyTurnFilter(int turn){
    this.turn = turn;
  }
  public boolean select(Perception ev){
    return ev.getLabel().equals("turn") &&
          (ev.intContent(0)>=turn);
  }
}
```

Fig. 10. Source code of the philosopher agent in the decentralized solution to the dining philosopher problem. The code of `getRightFork` and `releaseRightFork`, not shown, is analogous to, respectively, `getLeftFork` and `releaseLeftFork`.

in the `java.util.concurrent` library — much in the same way Java is less efficient than lower-level languages such as C. On the other hand, the framework allows for completely decoupling the definition of tasks, activities and operations as *logical* units of work from their actual executions by threads. The main benefit of this architectural choice – which is adopted also in other existing task-oriented concurrency frameworks, such as the Executor in the `java.util.concurrent` library – is scalability, making it possible to control the number of threads used to execute programs which may involve the creation of many, possibly complex agents and artifacts.

Currently the framework adopts a pool of threads for executing the two kinds of computational activities involved in the execution of a simpA program, i.e., agent activities and artifact operations. So, even if performance is not a primary concern of simpA, such decoupling makes it quite straightforward to investigate in the future more advanced policies to map activities and operations to threads.

3.5. Concurrency bugs and properties

Some features of the programming model have been thought to reduce the number of bugs that can occur in concurrent programs or to simplify their detection.

Concerning safety, on the artifact side no race conditions can occur in changing artifact state when multiple agents act upon the same artifact, given the computational model adopted for artifacts. Analogously, on the agent side no race conditions can occur in updating/inspecting the agenda in agents having multiple concurrent activities, given the atomicity of agent internal actions.

Then, we argue that the use of a declarative approach to specify – on the agent side – the relationships among the activities inside an agenda and – on the artifact side – the guard condition for operation step execution improves both the readability of the code and the detection of bugs, compared to approaches using directly lower-level mechanisms, such as condition variables in monitors. So, problems such as deadlock or livelock can still occur, for instance by defining wrong pre-conditions for todo in agents' agenda or wrong guards in operation steps; however, the reason for the problems should be easier to find by checking the content of the logic expressions, either manually or by means of proper automated tools. Actually, the price to pay for such a declarative approach is on the performance side, due to the overhead of the runtime evaluation of the logic expressions.

Finally, concerning agent–artifact interaction, the use of *timeouts* in actions such as `sense` and `use` makes it possible to program agents that do not starve in waiting for events that could possibly never occur, due to either bugs or bad use of artifacts.

4. Operational semantics

In this section we introduce the operational semantics for a significant subset of the simpA framework. The main aspects of interest that we model include (i) a basic object-oriented layer used to encode basic data types and algorithmic computation; (ii) an agent model focussing on activities, agendas, and sensors; and (iii) an artifact model focussing on operations, steps and event generation — aspects like timeouts, persistent todo, focussing and observable properties, are neglected for the sake of space and for keeping the formalization uniform and reasonably simple. The goal of the operational semantics provided is as a formal specification of simpA constructs that avoids ambiguities — a rather crucial tool in the context of complex, concurrent systems. In future works, this will pave the way towards establishing technical analysis results over simpA applications, for example, by equipping the language with a type system.

Following an established tradition of both the object-orientation and concurrency research contexts (see e.g. [22,23,55] for the former, and [6,36] for the latter), the operational semantics is developed on top of a calculus. That is, we introduce a rather small language that focuses on the main aspects of interest while abstracting away from other language mechanisms. With this approach, operational semantics amounts to the specification of a sort of virtual simpA interpreter (or engine), defined in a rigorous and uniform way in terms of rewriting rules. Technically, the formalization is structured in four parts, presented in the following subsections: first, an abstract syntax resembling the Java extension, listing the main operators and constructs considered; second, a runtime syntax, adding to the abstract syntax the runtime entities manipulated by the virtual simpA interpreter; third, evaluation contexts that define the places where the next expression has to be evaluated; and fourth, an operational semantics describing how runtime entities evolve over time.

4.1. Abstract syntax

This calculus heavily relies on the notations and syntactic conventions of the FJ (Featherweight Java) calculus, the de facto standard for modeling Java extensions [22].

We let metavariable `C` range over class names, `A` over artifact (template) names, `G` over agent (template) names, `f` over field names, `m` over method names, `a` over activity names, `o` over operation names and `x` over variables. `T` is used to represent types that can be used in the surface language, which ranges over `G`, `A`, `C`, and identifier `sns` (which stands for the type of sensors): note, however, that our core language is untyped, yet types are expressed for example in method and operation arguments to strengthen similarity with the actual language, though they would not be needed. Greek letters are used as metavariables ranging over identifiers for runtime instances of the above constructs: α for artifacts, γ for agents, ω for operations, and σ for sensors; metavariable ρ is additionally used for references α and γ. As usual in [22], we write "$\overline{\mathtt{e}}$" for "$\mathtt{e}_1, \ldots, \mathtt{e}_n$", "$\overline{\mathtt{T}}\ \overline{\mathtt{x}}$" for "$\mathtt{T}_1\,\mathtt{x}_1, \ldots, \mathtt{T}_n\,\mathtt{x}_n$", "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$"; for "$\mathtt{T}_1\,\mathtt{f}_1; \ldots; \mathtt{T}_n\,\mathtt{f}_n$", and the like ($n \geq 0$). The notation "$\overline{\mathtt{x}}$" is actually used for any metavariable `x`, and stands for a new metavariable over lists of elements of kind `x`; given list $\overline{\mathtt{x}}$, we denote by $\mathtt{x}_i$ its ith element. The notation $\mathtt{e}[\mathtt{e}'/\mathtt{x}]$ represents expression `e` where all occurrences of variable `x` are substituted with $\mathtt{e}'$, and similarly for the iterative version $\mathtt{e}[\overline{\mathtt{e}}/\overline{\mathtt{x}}]$; moreover, syntax $\overline{\mathtt{e}}[i \mapsto \mathtt{e}]$ represents list $\overline{\mathtt{e}}$ after substituting its ith element with `e`. The abstract syntax of the calculus is shown in Fig. 11.

A program is formed by a list of definitions `L`, of classes, agents and artifacts. A class definition includes its name `C`, a list of fields $\overline{\mathtt{f}}$ (each with its type $\mathtt{T}_i$), and a list of methods (class inheritance and constructors are not explicitly modeled). Methods have a return type `T`, a name `m` and arguments $\overline{\mathtt{T}}\ \overline{\mathtt{x}}$, and their body is just a return statement — as in FJ, classes are functional entities with no side effects. An agent (template) definition has a name `G` and a list of activities $\overline{\mathtt{Act}}$ representing its autonomous behavior. An activity has a name `a`, arguments $\overline{\mathtt{T}}\ \overline{\mathtt{x}}$, an optionally void agenda of activities, each with its own precondition $\mathtt{e}_i$, and a body formed by a statement (expressions and statements coincide). Note that while in simpA an activity has either a body (simple activity) or sub-activities (activity with agenda), we generalize by allowing activities with both a body and sub-activities — the body is executed only when the sub-activities completed. An artifact (template) definition has a name `A`, a list of fields representing its state, and a list of operations $\overline{\mathtt{Op}}$ and steps $\overline{\mathtt{Step}}$ representing the

L	::= `class C{`$\overline{\texttt{T f;}}$ $\overline{\texttt{Meth}}$ `}`	Class definition
	| `agent G{`$\overline{\texttt{Act}}$`}`	Agent definition
	| `artifact A{`$\overline{\texttt{T f;}}$ $\overline{\texttt{Op}}$ $\overline{\texttt{Step}}$`}`	Artifact definition
Meth	::= `T m(`$\overline{\texttt{T x}}$`){return e;}`	Method definition
Act	::= `activity a(`$\overline{\texttt{T x}}$`) agenda(`$\overline{\texttt{a e}}$`){b;}`	Activity definition
Op	::= `operation o(`$\overline{\texttt{T x}}$`) guard e{b;}`	Operation definition
Step	::= `step o(`$\overline{\texttt{T x}}$`) guard e{b;}`	Step definition
e, b	::= `x` | α | γ | ω | σ	Expressions: variable/id
	| `e.f` | `e.m(`$\overline{\texttt{e}}$`)`	field access/meth invocation
	| `e;e` | `new T(`$\overline{\texttt{e}}$`)`	sequence/instance creation
	| `.f` | `.f=e`	artifact-field access/update
	| `.#o(`$\overline{\texttt{e}}$`)` | `.>e`	next step/event generation
	| `e<-e..o(`$\overline{\texttt{e}}$`)` | `e.<`	operation use/event sensing
	| `read(e)` | `rem(e)` | `ins(e)`	memo operations

Fig. 11. Syntax.

v, w	::= ρ | σ | `new C(`$\overline{\texttt{v}}$`)`	Values
n	::= `a` | `n.a`	Nested activity
s	::= `o`$\langle$`e`$\rangle^*$`(`$\overline{\texttt{v}}$`)`	Pending step
M	::= 0	empty configuration
	| (M | M)	parallel composition
	| α = `A`*`(`$\overline{\texttt{v}}$`)[`$\overline{\omega^*}$`]`	artifact
	| γ = `G(`$\overline{\texttt{v}}$`)[`$\overline{\omega}$ $\overline{\sigma}$`]`	agent
	| $\sigma(\gamma)$ = `sns(`$\overline{\texttt{v}}$`)`	sensor
	| `n(`γ`)` = `[`$\overline{\texttt{a}^*}$`]{e:b}`*	activity
	| $\omega(\alpha)$ = γ`:o[`$\overline{\texttt{s}}$`]`$^*\langle$`e`$\rangle^*${b}*	operation

Fig. 12. Runtime constructs.

services it provides. Operations and steps have a name `o`, arguments $\overline{\texttt{T x}}$ and a guard expression `e`, and their body is a statement.

Expressions `e` (or `b`) are used both to model functional-like evaluation and step-by-step execution of instructions by a semicolon sequential composition operator—this choice ultimately results in a simpler yet expressive calculus. An expression can be a variable `x`, namely, an argument of the current method/activity/operation; inside classes, literal `this` is handled as a special kind of variable. The identifiers $\alpha, \gamma, \omega, \sigma$ do not belong to the surface language, but are generated as expressions are evaluated. Standard OO expressions are provided for accessing field `f` from receiver `e` (`e.f`), invoking method `m` from receiver `e` with arguments $\overline{\texttt{e}}$ (`e.m(`$\overline{\texttt{e}}$`)`), and creating an instance of type `T` with arguments $\overline{\texttt{e}}$ (`new T(`$\overline{\texttt{e}}$`)`). All the above expressions can be used in classes, agents, or artifacts. Other expressions are included to model primitives of the simpA framework as new language mechanisms. Inside artifacts, fields are accessed by syntax `.f` and are updated by `.f = e` (`e` evaluates to the new value), a new step is triggered by `.#o(`$\overline{\texttt{e}}$`)`, and an event with content `e` is generated by `.>e`. Inside agents, syntax σ`<-`α`..o(`$\overline{\texttt{e}}$`)` is used to use operation `o(`$\overline{\texttt{e}}$`)` on artifact α and to receive events on sensor σ; `e.<` is used to sense an event on sensor `e`; operators `read`, `rem` and `ins` are used to read, remove and insert memos. We abuse the operator "$\in$" to represent lookup functions, using it for syntactic structure inclusion: we write, for example, "`Meth` $\in$ `C`" to mean method that `Meth` is included in the definition of class `C`.

In developing the above syntax, we made a number of assumptions to trade off simplicity of the calculus and completeness of features with respect to simpA. Basically, the above syntax actually allows more programs than a compiler for the language should accept; for example, the next-step operator is in principle allowed in agents as well. We handle these kinds of check not syntactically, but as a typing or well-structuring problem, which we discuss in Section 4.6.

4.2. Configurations

While in standard functional (or OO) calculi operational semantics is defined as an evaluation transition over expressions, terminating in a value, in our context the state of computation is more involved, and needs the introduction of some runtime entities as shown in Fig. 12. As a syntactic convention, we adopt specific brackets for given kinds of elements: "()" for

$$
\begin{array}{lll}
\mathbb{E} ::= [\![]\!] \mid \mathbb{E};\mathtt{e} \mid \mathbb{E}.\mathtt{f} \mid \mathbb{E}.\mathtt{m}(\overline{\mathtt{e}}) \mid \mathtt{v}.\mathtt{m}(\overline{\mathtt{v}}\ \mathbb{E}\ \overline{\mathtt{e}}) & & \text{Expressions} \\
\quad \mid \mathtt{new\ T}(\overline{\mathtt{v}}\ \mathbb{E}\ \overline{\mathtt{e}}) \mid .\mathtt{f}{=}\mathbb{E} \mid .\#\mathtt{o}(\overline{\mathtt{v}}\ \mathbb{E}\ \overline{\mathtt{e}}) & & \\
\quad \mid \mathbb{E}\mathtt{<\text{-}e..o}(\overline{\mathtt{e}}) \mid \sigma\mathtt{<\text{-}}\mathbb{E}\mathtt{..o}(\overline{\mathtt{e}}) \mid \sigma\mathtt{<\text{-}}\alpha\mathtt{..o}(\overline{\mathtt{v}}\ \mathbb{E}\ \overline{\mathtt{e}}) & & \\
\mathbb{K} ::= \omega(\Downarrow) = \gamma:\mathtt{o}[\overline{\mathtt{s}^*}]\langle\mathbb{E}\rangle^{ev}\{\mathtt{b}\} \mid \omega(\Downarrow) = \gamma:\mathtt{o}[\overline{\mathtt{s}^*}]\langle\mathbf{t}\rangle\{\mathbb{E}\}^{ev} & & \text{Operations} \\
\quad \mid \omega(\Downarrow) = \gamma:\mathtt{o}[\overline{\mathtt{s}^*}\ \mathtt{o}\langle\mathbb{E}\rangle^{ev}(\overline{\mathtt{v}})]^{ev}\langle\mathtt{e}\rangle\{\mathtt{b}\} & & \text{Steps} \\
\quad \mid \mathtt{n}(\gamma) = [\overline{\mathtt{a}^*}]\{\mathbb{E}:\mathtt{b}\}^{ev} \mid \mathtt{n}(\gamma) = [\overline{\mathtt{a}^*}]\{\mathbf{t}:\mathbb{E}\}^{ev} & & \text{Activities}
\end{array}
$$

Fig. 13. Evaluation contexts.

arguments, "⟨⟩" for a guard or precondition, "{}" for a body, and "[]" for a list of pending elements (steps, operations, todos, and so on). Moreover, we label syntactic elements with state information, so as to keep track of state changes. Meta-variable "$*$" ranges over such labels, which can be possibly empty (typically meaning an idle state): for example, for an artifact we use *g* for guard evaluation, *e* for execution, and *s* for step-selection; for a bracket construct like {e:b}, label "*ev*" means that its content is under evaluation.

We first introduce the concept of a value v, representing as usual the possible outcome of the evaluation of an expression: values are either references (to artifacts, agents, or actions/sensors) or objects; namely, "new C($\overline{\mathtt{v}}$)" is an instance of class C where its fields are filled with values $\overline{\mathtt{v}}$, orderly. Hence, while agents and artifacts have an identifier, objects are expressed by their class and fields, that is, in a functional, stateless way like in FJ calculus. Since activities can be nested into agendas, a notation for nested activities n is introduced; for example, a.a′ means activity a′ within the agenda of top-level activity a. A pending step is represented by notation o⟨e⟩($\overline{\mathtt{v}}$), where o is the step name, e is the guard expression, and $\overline{\mathtt{v}}$ are the step arguments — label *ev* in a pending step means its guard can be evaluated.

The operational semantics will be described as a transition system over configurations, ranged over by M. A configuration is a multi-set of five kinds of elements, with composition operator " | ": (i) $\alpha = \mathtt{A}^*(\overline{\mathtt{v}})\ [\overline{\omega^*}]$ is the state of an artifact with identifier α, $\overline{\mathtt{v}}$ field values, and $\overline{\omega^*}$ pending operations; (ii) $\gamma = \mathtt{G}(\overline{\mathtt{v}})\ [\overline{\omega}\ \overline{\sigma}]$ is an agent with identifier γ, memos $\overline{\mathtt{v}}$, and where each couple $\omega_\mathtt{i}\ \sigma_\mathtt{i}$ is an association between operation and sensor; (iii) $\sigma(\gamma) = \mathtt{sns}(\overline{\mathtt{v}})$ is a sensor with identifier σ, belonging to agent γ, and with pending events $\overline{\mathtt{v}}$; (iv) $\mathtt{n}(\gamma) = [\overline{\mathtt{a}^*}]\{\mathtt{e}:\mathtt{b}\}^*$ is an activity with (nested) name n belonging to agent γ, with pending agenda $\overline{\mathtt{a}^*}$, precondition e and body b; and (v) $\omega(\alpha) = \gamma:\mathtt{o}[\overline{\mathtt{s}}]^*\langle\mathtt{e}\rangle\{\mathtt{b}\}$ is an operation with identifier ω, triggered by agent γ on artifact α, with name o, pending steps $\overline{\mathtt{s}}$, guard e, and body b. Configurations, lists inside [] brackets, and memos of an agent are supposed to be multi-sets, and we hence assume associativity and commutativity of their composition operator, whereas for other lists we only assume associativity.

4.3. Evaluation contexts

Expressions to be evaluated can appear in many places of a configuration; we therefore introduce evaluation contexts, which represent configurations ($\mathbb{K}$) and expressions ($\mathbb{E}$) with one hole in them, denoted by [[]] and representing the first place where evaluation can be applied [55,52]. Their syntax is expressed in Fig. 13. Boolean values are expressed by notations **t** and **f**, which are considered as objects "new True()" and "new False()" of library classes for booleans.

As an evaluation context $\mathbb{E}$ is basically an expression with one hole [[]] in it, syntax $\mathbb{E}[\![\mathtt{e}]\!]$ denotes the expression obtained by context $\mathbb{E}$ after substituting its hole with expression e. This syntactic construct is useful to identify what is the next sub-expression that needs to be evaluated, imposing a proper evaluation order by abstracting away from the other parts of the program where such a sub-expression is inserted. When an expression matches $\mathbb{E}[\![\mathtt{e}]\!]$, it means that e is the next sub-expression that needs to be evaluated: the idea is that when e evaluates to v, then the whole expression $\mathbb{E}[\![\mathtt{e}]\!]$ evaluates to $\mathbb{E}[\![\mathtt{v}]\!]$. The way the syntax of $\mathbb{E}$ is structured guarantees for example that a receiver is evaluated before the method arguments are evaluated, and that such arguments are evaluated from left to right, and so on.

Evaluation contexts $\mathbb{K}$ have another hole ⇓: $\mathbb{K}(\!|\rho|\!)[\![\mathtt{e}]\!]$ represents a configuration item belonging to agent or artifact ρ, where e is the next expression to evaluate. Inside operations, guards and bodies are executed when they are tagged by label *ev*; in steps, a guard step can be evaluated if the step is tagged *ev* and the whole set of steps is tagged *ev* (namely, the artifact is in selection state as will be discussed later); finally, in activities labelled *ev*, the precondition evaluates first, and if it reaches **t** then also the body can be executed. The actual impact of evaluation contexts on the operational semantics will be described in the following.

4.4. Operational semantics: execution of instructions

The operational semantics is defined as a non-deterministic transition system over configurations. For presentation purposes, in this section we first describe rules managing execution of instructions appearing inside evaluation contexts, which provide an account for the point-wise semantics of simpA constructs (access to fields, generation of events, triggering

$$\frac{\mathtt{M}' \longrightarrow \mathtt{M}''}{\mathtt{M} \mid \mathtt{M}' \longrightarrow \mathtt{M} \mid \mathtt{M}''} \quad [\text{PAR}]$$

$$\mathbb{K}(\!|\rho|\!)[\![\mathtt{v;e}]\!] \longrightarrow \mathbb{K}(\!|\rho|\!)[\![\mathtt{e}]\!] \quad [\text{SEQ}]$$

$$\frac{\overline{\mathtt{T}}\ \overline{\mathtt{f}} \in \mathtt{C}}{\mathbb{K}(\!|\rho|\!)[\![\mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{.f_i}]\!] \longrightarrow \mathbb{K}(\!|\rho|\!)[\![\mathtt{v_i}]\!]} \quad [\text{FIE}]$$

$$\frac{\mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\mathtt{\{return\ e;\}} \in \mathtt{C}}{\mathbb{K}(\!|\rho|\!)[\![\mathtt{new\ C}(\overline{\mathtt{v}})\mathtt{.m}(\overline{\mathtt{w}})]\!] \longrightarrow \mathbb{K}(\!|\rho|\!)[\![\mathtt{e[new\ C}(\overline{\mathtt{v}})\mathtt{/this][}\overline{\mathtt{w}}\mathtt{/}\overline{\mathtt{x}}\mathtt{]}]\!]} \quad [\text{INVK}]$$

Fig. 14. Evaluation of the OO layer.

of next steps, and so on), and then analyze the state transition of agents and artifacts, which emphasizes instead the concurrent behavior enacted in agents and artifacts — or rather, the agent and artifact abstract machine (selection of steps, disposal of completed activities, and so on).

General rules — Fig. 14 provides rules for evaluation of parallel composition, sequential composition, and typical object-oriented operations: it hence models aspects which are mostly orthogonal to agents and artifacts.[8] Rule [PAR] sets an interleaved parallelism model for this calculus, saying that any system subpart $\mathtt{M}'$ can spontaneously evolve to $\mathtt{M}''$. The only form of global awareness that we really require is due to predicate `fresh` (used in other rules) which is used to generate a globally new reference identifier, for agents, artifacts and operations — freshness of new identifiers is ensured in simpA at the implementation level through proper time stamps. Rule [SEQ] says that if we have a context $\mathbb{K}(\!|\rho|\!)[\![\mathtt{v;e}]\!]$, it means that the first statement of the block has already been executed, and hence we can move to $\mathbb{K}(\!|\rho|\!)[\![\mathtt{e}]\!]$. Rules [FIE] and [INVK] deal with field access and method invocation as for the standard object-oriented settings (they are in fact similar to the corresponding rules in FJ). In both cases the receiver is a fully evaluated expression of the kind "$\mathtt{new\ C}(\overline{\mathtt{v}})$", representing an object of class C with values $\overline{\mathtt{v}}$ in its fields. Accordingly, [FIE] retrieves the ith field, while [INVK] retrieves the expression body e after substituting `this` with "$\mathtt{new\ C}(\overline{\mathtt{v}})$", and formal parameters $\overline{\mathtt{x}}$ with actual arguments $\overline{\mathtt{w}}$.

Agent constructs – Fig. 15 provides the operational semantics of agents constructs, that is, instructions and operators that concern agents. Rule [AGN] handles creation of an agent with initial memos $\overline{\mathtt{v}}$, which yields a new reference γ, and creates a new agent instance and a `main` activity instance; similarly, rule [SNS] handles creation of a new sensor. Rule [RDMM] handles the operation for reading a memo matching a template given, which yields **f** if none is retrieved. Rules [RMMM, INSMM] handle operations for removing and inserting memos: note that differently from [RDMM] (which can be used also in preconditions), removing is blocked until a memo is found. Rule [USE] defines the semantics of operation use, which basically creates an operation instance: the operation identifier ω is returned, it is added to the artifact's set of idle (pending) operations, and a new association with the specified sensor σ is added to the agent instance. The sensing operation is handled by rule [PER], in which the first value v in the queue of the sensor – when available – is consumed and returned.

Artifact constructs — Fig. 16 similarly provides the operational semantics of basic instructions for artifacts, that is, instructions and operators that concern artifacts. Rule [ART] is used to create an artifact similarly to rule [AGN] for agents: a new operation instance for `init` is created and prepared for execution, linked to a fresh, dummy agent reference. Rule [GET] models artifact field access in a way similar to [FIE], while [SET] accordingly models the update. Rule [GEN] is used to handle event generation: it yields the event content v, and appends it to the queue of the proper sensor — namely, the sensor linked to the operation $\omega_\mathtt{i}$ as specified in the agent configuration. Rule [NEXT] handles triggering of the next step, which yields the artifact identifier and simply adds a new pending step to the operation instance.

4.5. Operational semantics: agent and artifact abstract machines

We now describe the state changes that occur in an agent while executing activities, and in an artifact while executing operations, which will emphasize details of the life-cycle of activities and operations. These are given in terms of transition rules as before (though they do not work on evaluation contexts, but rather on whole configurations). Additionally, we show Petri net schemata that help in visualizing the fundamentals of this part of the operation semantics.

[8] In principle we could have used a modular approach as in [37] to rely on a more abstract domain of values, but we believe that our approach is still simpler for it requires only three additional rules for handling the object-oriented layer.

$$\frac{\gamma \text{ fresh} \quad \texttt{activity main() agenda}(\overline{\texttt{a}}\ \overline{\texttt{e}})\{\texttt{b;}\} \in \texttt{G}}{\mathbb{K}(\!|\rho|\!)[\![\texttt{new G}(\overline{\texttt{v}})]\!] \longrightarrow \mathbb{K}(\!|\rho|\!)[\![\gamma]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}})[] \mid \texttt{main}(\gamma) = [\overline{\texttt{a}}]\{\mathbf{t}\texttt{:b}\}^{ev}} \quad [\text{AGN}]$$

$$\frac{\sigma \text{ fresh}}{\mathbb{K}(\!|\gamma|\!)[\![\texttt{new sns()}]\!] \longrightarrow \mathbb{K}(\!|\gamma|\!)[\![\sigma]\!] \mid \sigma(\gamma) = \texttt{sns()}} \quad [\text{SNS}]$$

$$\frac{(\texttt{w=v if v matches v}') \quad \text{or} \quad (\texttt{w=}\mathbf{f}\text{ otherwise})}{\mathbb{K}(\!|\gamma|\!)[\![\texttt{read(v}')]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}}\ \texttt{v})[\overline{\omega}\ \overline{\sigma}] \longrightarrow \mathbb{K}(\!|\gamma|\!)[\![\texttt{w}]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}}\ \texttt{v})[\overline{\omega}\ \overline{\sigma}]} \quad [\text{RDMM}]$$

$$\frac{\texttt{v matches v}'}{\mathbb{K}(\!|\gamma|\!)[\![\texttt{rem(v}')]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}}\ \texttt{v})[\overline{\omega}\ \overline{\sigma}] \longrightarrow \mathbb{K}(\!|\gamma|\!)[\![\texttt{v}]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}]} \quad [\text{RMMM}]$$

$$\mathbb{K}(\!|\gamma|\!)[\![\texttt{ins(v)}]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}] \longrightarrow \mathbb{K}(\!|\gamma|\!)[\![\texttt{v}]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}}\ \texttt{v})[\overline{\omega}\ \overline{\sigma}] \quad [\text{INSMM}]$$

$$\frac{\omega \text{ fresh} \quad \texttt{operation o}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{guard e}\{\texttt{b}\} \in \texttt{A}}{\begin{array}{l}\mathbb{K}(\!|\gamma|\!)[\![\sigma\texttt{<-}\alpha\texttt{..o}(\overline{\texttt{v}})]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}'})[\overline{\omega}\ \overline{\sigma}] \mid \alpha = \texttt{A}^*(\overline{\texttt{w}})[\overline{\omega'}] \longrightarrow \\ \mathbb{K}(\!|\gamma|\!)[\![\omega]\!] \mid \gamma = \texttt{G}(\overline{\texttt{v}'})[\overline{\omega}\ \overline{\sigma},\ \omega\ \sigma] \mid \alpha = \texttt{A}^*(\overline{\texttt{w}})[\omega,\overline{\omega'}] \mid \\ \omega(\alpha) = \gamma\texttt{:o}[]\langle\texttt{e}\rangle\{\texttt{b}[\overline{\texttt{v}}/\overline{\texttt{x}}]\}\end{array}} \quad [\text{USE}]$$

$$\mathbb{K}(\!|\gamma|\!)[\![\sigma\texttt{.<}]\!] \mid \sigma(\gamma) = \texttt{sns(v,}\overline{\texttt{v}}\texttt{)} \longrightarrow \mathbb{K}(\!|\gamma|\!)[\![\texttt{v}]\!] \mid \sigma(\gamma) = \texttt{sns}(\overline{\texttt{v}}) \quad [\text{PER}]$$

Fig. 15. Evaluation of basic agent constructs.

$$\frac{\alpha, \omega, \gamma \text{ fresh} \quad \texttt{operation init() guard true}\{\texttt{b;}\} \in \texttt{A}}{\begin{array}{l}\mathbb{K}(\!|\rho|\!)[\![\texttt{new A}(\overline{\texttt{v}})]\!] \longrightarrow \\ \mathbb{K}(\!|\rho|\!)[\![\alpha]\!] \mid \alpha = \texttt{A}^e(\overline{\texttt{v}})[\omega^e] \mid \omega(\alpha) = \gamma\texttt{:init}[]\langle\mathbf{t}\rangle\{\texttt{b}\}^{ev}\end{array}} \quad [\text{ART}]$$

$$\frac{\overline{\texttt{T}}\ \overline{\texttt{f}} \in \texttt{A}}{\mathbb{K}(\!|\alpha|\!)[\![\texttt{.f}_\texttt{i}]\!] \mid \alpha = \texttt{A}^*(\overline{\texttt{v}})[\overline{\omega^*}] \longrightarrow \mathbb{K}(\!|\alpha|\!)[\![\texttt{v}_\texttt{i}]\!] \mid \alpha = \texttt{A}^*(\overline{\texttt{v}})[\overline{\omega^*}]} \quad [\text{GET}]$$

$$\frac{\overline{\texttt{T}}\ \overline{\texttt{f}} \in \texttt{A} \quad \overline{\texttt{w}} = \overline{\texttt{v}}[\texttt{i} \mapsto \texttt{w}]}{\mathbb{K}(\!|\alpha|\!)[\![\texttt{.f}_\texttt{i}\texttt{=w}]\!] \mid \alpha = \texttt{A}^e(\overline{\texttt{v}})[\overline{\omega^*}] \longrightarrow \mathbb{K}(\!|\alpha|\!)[\![\texttt{w}]\!] \mid \alpha = \texttt{A}^e(\overline{\texttt{w}})[\overline{\omega^*}]} \quad [\text{SET}]$$

$$\begin{array}{l}\omega_\texttt{i}(\alpha) = \gamma\texttt{:o}[\overline{\texttt{s}}]\langle\mathbf{t}\rangle\{\mathbb{E}[\![\texttt{.>v}]\!]\}^{ev} \mid \gamma = \texttt{G}(\overline{\texttt{v}'})[\overline{\omega}\ \overline{\sigma}] \mid \sigma_\texttt{i}(\gamma) = \texttt{sns}(\overline{\texttt{v}}) \longrightarrow \\ \omega_\texttt{i}(\alpha) = \gamma\texttt{:o}[\overline{\texttt{s}}]\langle\mathbf{t}\rangle\{\mathbb{E}[\![\texttt{v}]\!]\}^{ev} \mid \gamma = \texttt{G}(\overline{\texttt{v}'})[\overline{\omega}\ \overline{\sigma}] \mid \sigma_\texttt{i}(\gamma) = \texttt{sns}(\overline{\texttt{v}},\texttt{v})\end{array} \quad [\text{GEN}]$$

$$\frac{\texttt{step o}'(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{guard e}\{\texttt{...}\} \in \texttt{A}}{\begin{array}{l}\omega(\alpha) = \gamma\texttt{:o}[\overline{\texttt{s}}]\langle\mathbf{t}\rangle\{\mathbb{E}[\![\texttt{.\#>o}'(\overline{\texttt{v}})]\!]\}^{ev} \mid \alpha = \texttt{A}^e(\overline{\texttt{w}})[\overline{\omega^*}] \longrightarrow \\ \omega(\alpha) = \gamma\texttt{:o}[\overline{\texttt{s}}\ \texttt{o}'\langle\texttt{e}\rangle(\overline{\texttt{v}})]\langle\mathbf{t}\rangle\{\mathbb{E}[\![\alpha]\!]\}^{ev} \mid \alpha = \texttt{A}^e(\overline{\texttt{w}})[\overline{\omega^*}]\end{array}} \quad [\text{NEXT}]$$

Fig. 16. Evaluation of basic artifact constructs.

$$\frac{\texttt{activity a() agenda}(\overline{\texttt{a}}\ \overline{\texttt{e}})\{\ldots\} \in \texttt{G} \quad \texttt{activity a}_\texttt{i}\texttt{() agenda}(\overline{\texttt{a}}'\ \overline{\texttt{e}}')\{\texttt{b}'\} \in \texttt{G}}{\begin{array}{l}\gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}]\,|\,\texttt{n.a}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}]\{\texttt{e:b}\} \longrightarrow \\ \gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}]\,|\,\texttt{n.a}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}^f]\{\texttt{e:b}\}\,|\,\texttt{n.a.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}}']\{\texttt{e}_\texttt{i}\texttt{:b}'\}\end{array}} \quad [\text{SCHD}]$$

$$\frac{}{\begin{array}{l}\texttt{n}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}^f]\{\texttt{e:b}\}\,|\,\texttt{n.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}'^c}]\{\texttt{e}'\texttt{:b}'\} \longrightarrow \\ \texttt{n}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}^e]\{\texttt{e:b}\}\,|\,\texttt{n.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}'^c}]\{\texttt{e}'\texttt{:b}'\}^{ev}\end{array}} \quad [\text{EXEC}]$$

$$\frac{\texttt{activity a() agenda}(\overline{\texttt{a}}\ \overline{\texttt{e}})\{\ldots\} \in \texttt{G}}{\gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}]\,|\,\texttt{n.a.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}^*}]\{\texttt{f:b}\}^{ev} \longrightarrow \gamma = \texttt{G}(\overline{\texttt{v}})[\overline{\omega}\ \overline{\sigma}]\,|\,\texttt{n.a.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}^*}]\{\texttt{e}_\texttt{i}\texttt{:b}\}^{ev}} \quad [\text{PREC}]$$

$$\frac{}{\begin{array}{l}\texttt{n}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}^e]\{\texttt{e:b}\}\,|\,\texttt{n.a}_\texttt{i}(\gamma) = [\overline{\texttt{a}'^c}]\{\texttt{t:v}\}^{ev} \longrightarrow \\ \texttt{n}(\gamma) = [\overline{\texttt{a}^*}\ \texttt{a}_\texttt{i}^c]\{\texttt{e:b}\}\end{array}} \quad [\text{DISP}]$$

Fig. 17. Agent state changes.

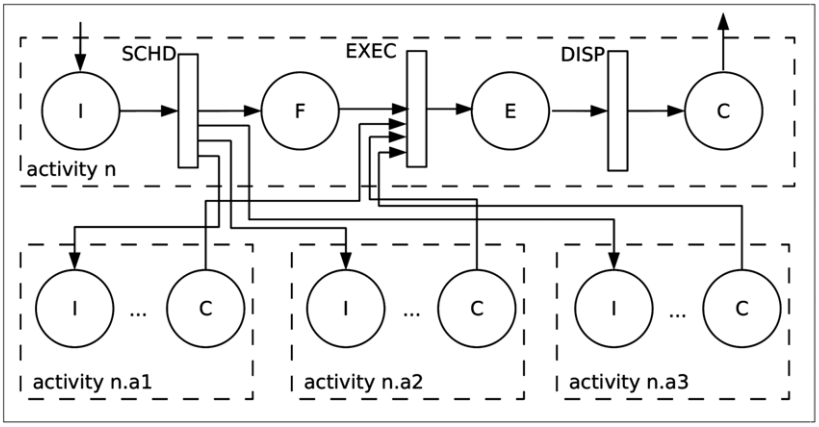

Fig. 18. Petri net schema describing the life-cycle of an activity n with three sub-activities $a1$, $a2$, $a3$.

Agent abstract machine — Fig. 17 provides the operational semantics of agent behavior, and Fig. 18 accordingly describes the main aspects of activities life-cycle in terms of Petri nets. Given activity $\texttt{n.a}$, rule [SCHD] takes at any time an idle todo $\texttt{a}_\texttt{i}$ and schedules it: a proper activity instance is created, and the state of $\texttt{a}_\texttt{i}$ moves to $\texttt{a}_\texttt{i}^f$ in $\texttt{n.a}$'s agenda (f stands for "firing"). Rule [EXEC] states that when all sub-activities of $\texttt{n.a}_\texttt{i}$ are labelled c, meaning "completed", then this activity can execute: this is obtained by marking "ev" the guarded body $\{\texttt{e}':\texttt{b}'\}$, which makes this activity matching an execution context $\mathbb{K}$. If, during execution of such activity, the guards would evaluate to false, by rule [PREC] we reset the guard to its initial value, so it can be evaluated again. Rule [DISP] handles the disposal of an activity, which occurs when the agenda is completed and the body is completely evaluated: most importantly, this moves the state of this activity to completed (label c) in its parent's agenda.

Fig. 18 shows an equivalent Petri net description, which is compositional since each dashed box represents one activity. The activity transits across states i (idle), f (firing), e (executed) and c (completed): transition SCHD puts tokens into the places of the sub-activities in order to initiate them, and EXEC starts only when such sub-activities are completed.

Artifact abstract machine — Fig. 19 provides the operational semantics of artifact behavior, and Fig. 20 accordingly describes the main aspects of an operation's life-cycle in terms of Petri nets. When an artifact is in an idle state, by rule [GRD] a pending operation is selected for guard evaluation. Context $\mathbb{G}$ allows this evaluation, resulting in one of the two following cases: if the evaluation is negative, rule [FLS] labels the operation as "completed" (preventing re-evaluation); if the evaluation is positive, rule [GO] moves the artifact and operation to the execution state, cleaning all "completed" labels of current operations. Context $\mathbb{B}$ now causes the evaluation of the body until it becomes a value, namely, its execution is over. Rules [END] and [STOP] move the artifact to the selection state: the former drops the operation if no pending steps exist; otherwise, the latter

$$\alpha = \mathtt{A}(\overline{\mathtt{w}})[\omega\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathtt{e}\rangle\{\mathtt{b}\} \longrightarrow \quad \text{[GRD]}$$
$$\alpha = \mathtt{A}^g(\overline{\mathtt{w}})[\omega^g\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathtt{e}\rangle^{ev}\{\mathtt{b}\}$$

$$\frac{\texttt{operation o}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \texttt{guard e}\{\mathtt{b}\} \in \mathtt{A}}{\alpha = \mathtt{A}^g(\overline{\mathtt{w}})[\omega^g\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathbf{f}\rangle^{ev}\{\mathtt{b}\} \longrightarrow} \quad \text{[FLS]}$$
$$\alpha = \mathtt{A}(\overline{\mathtt{w}})[\omega^c\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathtt{e}\rangle\{\mathtt{b}\}$$

$$\alpha = \mathtt{A}^g(\overline{\mathtt{w}})[\omega^g\ \overline{\omega_0}\ \overline{\omega_1}^c\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathbf{t}\rangle^{ev}\{\mathtt{b}\} \mid \mathbb{K}(\!|\gamma|\!)[\![\omega]\!] \longrightarrow \quad \text{[GO]}$$
$$\alpha = \mathtt{A}^e(\overline{\mathtt{w}})[\omega^e\ \overline{\omega_0}\ \overline{\omega_1}\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathbf{t}\rangle\{\mathtt{b}\}^{ev} \mid \mathbb{K}(\!|\gamma|\!)[\![\mathbf{t}]\!]$$

$$\alpha = \mathtt{A}^e(\overline{\mathtt{w}})[\omega^e\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathbf{t}\rangle\{\mathtt{v}\}^{ev} \longrightarrow \alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\overline{\omega^*}] \quad \text{[END]}$$

$$\alpha = \mathtt{A}^e(\overline{\mathtt{w}})[\omega^e\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\mathtt{s}\ \overline{\mathtt{s}}]\langle \mathbf{t}\rangle\{\mathtt{v}\}^{ev} \longrightarrow \quad \text{[STOP]}$$
$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^p\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\mathtt{s}\ \overline{\mathtt{s}}]\langle \mathbf{t}\rangle\{\mathtt{v}\}$$

$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\overline{\omega_0}\ \overline{\omega_1}^b] \longrightarrow \alpha = \mathtt{A}(\overline{\mathtt{w}})[\overline{\omega_0}\ \overline{\omega_1}^p] \quad \text{[STUCK]}$$

$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^p\ \overline{\omega_0}\ \overline{\omega_1}^b\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}}]\langle \mathbf{t}\rangle\{\mathtt{v}\} \longrightarrow \quad \text{[S-PICK]}$$
$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^s\ \overline{\omega_0}\ \overline{\omega_1}^b\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}}]^{ev}\langle \mathbf{t}\rangle\{\mathtt{v}\}$$

$$\frac{\texttt{step o}'(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \texttt{guard e}\{\mathtt{b}\} \in \mathtt{A}}{\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^s\ \overline{\omega_0}\ \overline{\omega_1}^b\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}^*}\ \mathtt{o}'\langle \mathbf{t}\rangle(\overline{\mathtt{v}})]^{ev}\langle \mathbf{t}\rangle\{\mathtt{v}\} \longrightarrow} \quad \text{[S-GO]}$$
$$\alpha = \mathtt{A}^e(\overline{\mathtt{w}})[\omega^e\ \overline{\omega_0}\ \overline{\omega_1}^p\ \overline{\omega_2}^p] \mid \omega(\alpha) = \gamma : \mathtt{o}[]\langle \mathbf{t}\rangle\{\mathtt{b}\}^{ev}$$

$$\frac{\texttt{step o}'(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \texttt{guard e}\{\mathtt{b}\} \in \mathtt{A}}{\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}^*}\ \mathtt{o}'\langle \mathbf{f}\rangle(\overline{\mathtt{v}})]^{ev}\langle \mathbf{t}\rangle\{\mathtt{v}\} \longrightarrow} \quad \text{[S-NO]}$$
$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}^*}\ \mathtt{o}'\langle \mathtt{e}\rangle(\overline{\mathtt{v}})^c]^{ev}\langle \mathbf{t}\rangle\{\mathtt{v}\}$$

$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^s\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}^c}]^{ev}\langle \mathbf{t}\rangle\{\mathtt{v}\} \longrightarrow \quad \text{[S-ALL]}$$
$$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})[\omega^b\ \overline{\omega^*}] \mid \omega(\alpha) = \gamma : \mathtt{o}[\overline{\mathtt{s}}]\langle \mathbf{t}\rangle\{\mathtt{v}\}$$

Fig. 19. Artifact: Rules for state changes.

fires, which moves the operation to the "pending" state. After rule [END], all pending operations are evaluated for selection, by evaluating all the guards of their pending steps: if none exists or they are all negatively evaluated (in which case pending operations are of the kind $\overline{\omega^c}$) rule [STUCK] moves the artifact back to the idle state. Otherwise, rule [S-PICK] selects a pending operation and moves it to the step selection state, where by context $\mathbb{G}$ the guards of pending steps are evaluated. We can now have three cases: [S-GO], if a guard evaluates to true, artifact and operation are moved to execution, other pending steps are discarded, and the evaluation state for other operations is cleaned (namely, blocked operations become pending again); [S-NO], if a guard evaluates to false, the state of the pending event is marked completed (i.e., negatively evaluated), and the original guard is restored for future re-evaluation; finally, [S-ALL], if all steps are found as negatively evaluated, the overall operation is marked as negatively evaluated — hence either rule [S-PICK] finds another pending operation, or [S-STUCK] moves the artifact state back to idle.

The Petri nets in Fig. 20 describe this behavior in a pictorial way: the top part and the bottom are to be joined to obtain the exact artifact/operation evolution (e.g. transition [FLS] has incoming arcs from G, A^I and outgoing arcs to C, A^I) — they have been kept separated for clarity. Double-arrowed arcs are used to represent so-called transfer arcs [15]: when one of

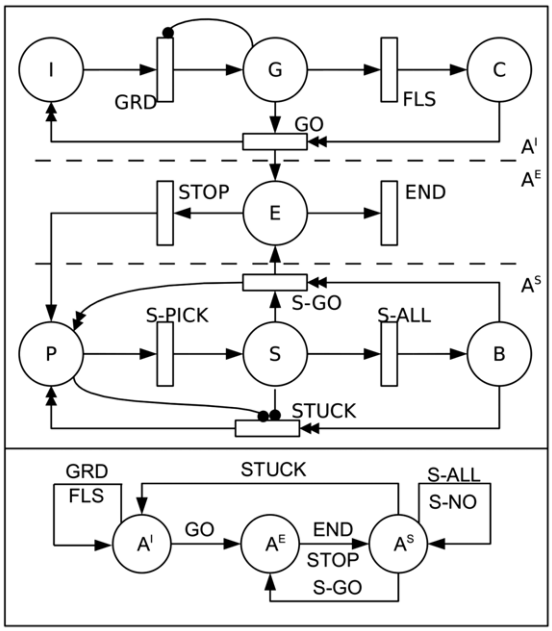

Fig. 20. Petri nets describing artifact life-cycle (bottom) and operations life-cycle (top); note that (i) the overall net is the union of the two nets; (ii) the three areas on top indicate transitions enabled by artifact states A^I, A^E, A^S; and (iii) double-arrowed edges represent transfer arcs.

such arcs connects place P with transition T and another such arc transition T with place P', then as T fires all tokens in P get transferred to P' [15] (e.g., firing transition GO, besides moving one token from G to E, also transfers all tokens from C to I).

4.6. Well-formedness and properties

In order to provide a more abstract characterization of the small-step operational semantics we here state some properties of well-formed programs, which will also help unveiling some properties of simpA programs.

Well-formedness — A simpA program is said to be well formed when the following conditions hold.

- GuardWF — guards of operations and steps include only OO constructs, and accesses of artifact fields (namely constructs handled by rules [INVK, FIE, SEQ, GET]);
- OpWF — bodies of operations and steps additionally include assignments of artifact fields, creation of new agents/artifacts, generation of events and triggering of new steps (see constructs handled in [SET, ART, AGN, GEN, NEXT]);
- PrecWF — preconditions of agent activities include only OO constructs, and operations for reading memos (see constructs handled in [INVK, FIE, SEQ, GET, RDMM]);
- ActWF — bodies of agent activities additionally include operations for removing/inserting memos, sensing events, using artifacts, and creating agents/artifacts/sensors (see constructs handled in [RMMM, INSMM, PER, USE, ART, AGN, SNS]);
- OOCheck — the correctness of OO constructs is checked using standard techniques as of FJ [22];
- OpCheck — the correctness of artifact operations (field access and update, and triggering of a new step) is checked against the definition of the artifact in which they occur (they resemble field access/update and method invocation in OO languages);
- NewCheck — the correctness of artifact and agent creation is checked against their definition (they resemble class instantiation in OO languages).

Note that in this definition we do not check use operations — though their checking is similar to method invocation. This is because we provide no guarantee of their execution anyway, since the precondition of an operation may turn out to be indefinitely evaluated to false, in which case the calling agent remains blocked just as if the use operation were not well formed (e.g. the operation name does not match any artifact operation).

A valid initial state for a system configuration is formed by the execution of an agent's `main` activity. More precisely, given an agent G, let $\overline{\mathtt{v}}$ be a set of fully evaluated arguments, $\overline{\mathtt{a}}$ its set of sub-activities, and b the body of its main activity; then the initial state generated by G is

$$\gamma \;=\; \mathtt{G}(\overline{\mathtt{v}})[]\,|\,\mathtt{main}(\gamma) \;=\; [\overline{\mathtt{a}}]\{\mathbf{t}\!:\!\mathtt{b}\}^{ev}.$$

Termination & progress — Given a well-formed simpA program and a valid initial state S_0, any computation path from S is of one of the following four kinds.

- Evaluation loop — The computation path is infinite, due to a loop during the evaluation of a guard, precondition, operation body, or activity body, namely, a recursive pattern of method calls. A restrictive compiler for simpA should statically prevent this situation, with techniques likes those in [51].
- Next-step loop — The computation path is infinite, due to a loop in the triggering of next steps, namely, steps continuously trigger each other as in recursive method calls. This behavior may either be prevented statically as in the case above, or be allowed so as to model those cases where artifacts continuously generate events (e.g. as in timer artifacts).
- Termination — The computation path reaches a correct termination of agent behavior, namely, all agent activities have been executed (they have been disposed), and all artifacts are idle and their operations (if any) are in "pending" state. Formally, one such state has only elements of the kind $\gamma = \mathtt{G}(\overline{\mathtt{v}}\ \mathtt{v})\,[\overline{\omega}]$, $\sigma(\gamma) = \mathtt{sns}(\overline{\mathtt{v}})$, $\alpha = \mathtt{A}(\overline{\mathtt{w}})\,[\overline{\omega^p}]$, plus terms for operations in $\overline{\omega}$. We allow pending operations for our notion of correct termination concerns the proactive behavior of agents only.
- Deadlock — The computation path reaches a deadlock of agent behavior, namely, some agent activity is blocked on perceiving an event, using an artifact operation, or reading a memo. Formally, one such state has at least one element of the kind $\mathbb{K}(\!|\gamma|\!)[\![\sigma\ .<]\!]$, $\mathbb{K}(\!|\gamma|\!)[\![\omega]\!]$, and $\mathbb{K}(\!|\gamma|\!)[\![\mathtt{rem(v)}]\!]$.

Concurrency properties — Given a well-formed simpA program, and any computation path from a valid initial state S_0, the following properties hold.

- Activity precondition — An activity body is never executed until its sub-activities complete and its precondition is positively evaluated. Formally, in any state S of the path we have

  $$\mathtt{n}(\gamma) = [\overline{\mathtt{a}^*}]\{\mathbf{t}\!:\!\mathtt{b}\}^{ev} \in S \Rightarrow \overline{\mathtt{a}^*} = \overline{\mathtt{a}^c}.$$

  This is a consequence of [EXEC] transition in Fig. 18, which requires completion of sub-activities.
- Activity parallelism — Two (or more) activities of one agent may concurrently be in evaluation state. Formally, for some initial state and some state S in a computation path we may have

  $$\mathtt{n}(\gamma) = [\overline{\mathtt{a}^c}]\{\mathbf{t}\!:\!\mathtt{b}\}^{ev} \mid \mathtt{n}'(\gamma) = [\overline{\mathtt{a}'^c}]\{\mathbf{t}\!:\!\mathtt{b}'\}^{ev} \in S.$$

  This is a consequence of the [SCHD] transition in Fig. 18, where all sub-activities are independently fired.
- Steps invariant — An artifact in idle state is such that no guard step positively evaluates. Formally, if a state S is such that

  $$\alpha = \mathtt{A}(\overline{\mathtt{w}})\,[\overline{\omega^*}] \mid \omega(\alpha) = \gamma\!:\!\mathtt{o}[\overline{\mathtt{s}}\ \ \mathtt{o}'\langle\mathtt{e}\rangle(\overline{\mathtt{v}})\,]\langle\mathbf{t}\rangle\{\mathtt{v}\} \in S$$

  then each computation from the following state

  $$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})\,[\omega^s] \mid \omega(\alpha) = \gamma\!:\!\mathtt{o}[\mathtt{o}'\langle\mathtt{e}\rangle^{ev}(\overline{\mathtt{v}})\,]^{ev}\langle\mathbf{t}\rangle\{\mathtt{v}\}$$

  necessarily reaches state

  $$\alpha = \mathtt{A}^s(\overline{\mathtt{w}})\,[\omega^s] \mid \omega(\alpha) = \gamma\!:\!\mathtt{o}[\mathtt{o}'\langle\mathbf{f}\rangle^{ev}(\overline{\mathtt{v}})\,]^{ev}\langle\mathbf{t}\rangle\{\mathtt{v}\}.$$

  This is a consequence of the net in Fig. 20, where the transition [STUCK] (moving artifact state to idle) is enabled only if all pending operations reached the blocked state (which happens if all their guard steps were evaluated to false); additionally, while in the idle state, no changes to the artifact state can occur, and hence preconditions do not change their value.
- Operation isolation — Only one step is in execution at a time. Formally, in any state S we have

  $$\omega(\alpha) = \gamma\!:\!\mathtt{o}[\overline{\mathtt{s}}]\langle\mathbf{t}\rangle\{\mathtt{b}\}^{ev} \mid \omega'(\alpha) = \gamma'\!:\!\mathtt{o}'[\overline{\mathtt{s}'}]\langle\mathbf{t}\rangle\{\mathtt{b}'\}^{ev} \notin S \quad (\gamma \neq \gamma').$$

  This is a consequence of the fact that in the net of Fig. 20 state E is easily shown to be 1-bounded.

5. Related works

Existing research on artifacts — The simpA model and technology are strictly related to the research work on artifacts for MAS [46], and the CArtAgO infrastructure [43]. While simpA introduces a specific programming model for programming agents and artifacts, namely to structure concurrent applications on top of Java, CArtAgO is an infrastructure focussed solely on developing artifacts. As such, it provides an API that agent designers can use to create agents interacting with artifacts. CArtAgO is designed to be integrated with heterogeneous agent platforms, including fully fledged cognitive ones, with the goal of extending any agent technology with a support to artifact-based environments.

The artifact abstraction at the core of simpA and CArtAgO is a generalization of *coordination artifacts* — i.e., artifacts encapsulating coordination functionalities, introduced in [40]. In A&A, artifacts are the basic building blocks that can be used to engineer the working environments where agents are situated: the agent environment plays a fundamental role in MAS engineering, acting as first-class entity encapsulating key responsibilities (functionality, services). This perspective is explored in several research works that have appeared recently in MAS literature: a survey can be found in [54]. By providing

a general-purpose programming model for artifacts, simpA gives the possibility to program any kind of coordination artifacts, from simple synchronizers (such as latch and barriers) to more complex ones, such as tuple spaces [18], tuple centres [38], and even workflow engines [53].

Coordination models and technology — Coordination models like programmable coordination media [14] and tuple centres [38] have been a main source of inspiration for devising the notion of coordination artifact in multi-agent systems. Actually, the programming model adopted for artifacts in general has been conceived to ease the implementation of coordination media, like the simple tuple space shown in Fig. 8, the synchronization barrier and the bounded buffer shown in Fig. 4. In particular, structured operations – composed by multiple guarded atomic operation steps – have been introduced to ease the design and implementation of coordinating behaviors involving the synchronization among actions of different agents, interacting through the medium, possibly including also information exchange. This model has been inspired – in particular – by *reactions* introduced by the tuple centre model (and the ReSpecT language, in particular) as a means to program the behavior of tuple spaces. Differently from tuple spaces and tuple centres, the adoption of observable events to model the asynchronous information flow from artifacts to using/observing agents also makes the approach quite effective for implementing event-driven coordination models, like those derived from the IWIM model [4].

Agent features in OO frameworks — simpA is not the first approach providing an agent-oriented abstraction layer on top of Java. JADE [7] is probably the most known Java-based agent platform. JADE provides a general-purpose middleware for developing peer-to-peer distributed agent-based applications, complying with the FIPA standardization aim.[9]

A main conceptual and practical difference between simpA and JADE concerns the high-level first-class abstractions adopted to organize a software system: JADE provides agents interacting by means of FIPA ACL (agent communication language), while simpA provides agents and artifacts — so in JADE every component of a multi-agent system must be modelled as an agent and there are no explicit first-class abstractions to model the environment. Furthermore, JADE adopts a *behavior-based* model for programming agents. From this point of view, activities in simpA are similar to behaviors in JADE, with the main difference that in simpA the definition of structured activities composed of sub-activities is done declaratively by defining the activity agenda, while in JADE this is done operationally, by creating and composing objects of specific classes. In this perspective it is worth remarking that a recent extension of JADE called WADE [11] has independently adopted an approach more similar to simpA activities, by introducing a class of agents (`WorkflowEngineAgent`) which are able to execute workflows of *activities*.

Besides JADE, other well-known agent-oriented platforms have been developed as an extension of the Java platform: we cite here JACK,[10] and JADEX [41], which differently from JADE and simpA provide a first-class support for programming *intelligent* agents, based on the BDI architecture (Beliefs–Desires–Intentions) and the FIPA standards. These approaches – as with most other cognitive agent programming platforms – are typically targeted to the engineering of distributed intelligent systems for complex application domains, not for concurrent programming in general as simpA.

Concurrency in OO languages — The extension of the OO paradigm toward concurrency – namely, object-oriented concurrent programming (OOCP) – has been (and indeed still is) one of the most important and challenging themes in the OO research. Accordingly, a large number of theoretical results and approaches have been proposed since the beginning of the 1980's, surveyed by works such as [10,56,3]. Among the main examples, *actors* [1] and *active objects* [29] have been the root of entire families of proposals. Among the most recent ones, actors are the reference concurrency model implemented in Erlang [5,28] (where they are called *processes*) and Scala [20].

The approach proposed in this paper shares the aim of actor and active objects approaches, i.e., to introduce a general-purpose abstraction layer to simplify the development of concurrent applications. Differently from actor-based approaches, in simpA also the passive components of the system are modeled as first-class entities (the artifacts), alongside the active parts (actors in actor-based systems). Consequently, the interaction model between agents and artifacts, based on use and observation, is very different from the interaction model among actors, based on asynchronous message passing — adopted also in simpA for inter-agent communication. Being based essentially on actions and events, this interaction model is quite different also from the synchronous and asynchronous message passing style adopted in active objects. Then, actor-based approaches typically do not provide specific constructs to structure the active behavior of an actor, besides the *become*-like primitives: this contrasts the activity-based model of simpA, which allows for structuring the behavior in a hierarchy of activities, possibly specifying declaratively the dependencies among the tasks to be executed as defined in the activity agenda.

Actually, besides using asynchronous message passing, the basic actor model has been extended so as to have a richer support for coordination and interaction management. Two main examples are the research work on *synchronizers* [17] and *actor spaces* [2]. In both cases, further abstractions are introduced besides actors, functioning as coordination media that make it possible to constrain and shape actor message-based communication. In simpA, these functionalities are provided by coordination artifacts, these being artifacts specifically programmed to encapsulate and enact coordinating functionalities for the agents sharing and using them.

On the other hand, artifacts resemble *monitors*, with some important differences. Analogously to monitors, artifacts enforce mutual exclusion in the execution of operation (steps). Differently from them, operations – once triggered – are

[9] http://www.fipa.org.

[10] http://www.agent-software.com.

executed asynchronously with respect to the (agent) invoker, and guards are adopted to enable/disable/synchronize the execution of operations (steps) instead of condition variables. Also, differently from monitors, there are no return values or output parameters in operations, since the interaction model is based on events, and a part of the state (observable properties) can be directly observed without invoking operations.

Finally, among the many approaches extending OO towards concurrency we mention here SCOOP [34] extending the Eiffel language, and the more recent Polyphonic C# [8], introducing a very elegant support – grounded on Join Calculus theory – for the synchronization and coordination of multi-threaded applications. An approach that is similarly based on the Join Calculus [16] is JOIN JAVA [25]. Other approaches include JEEG [35], which is able to express synchronization conditions on an object by *linear temporal logic constraints* involving the value of fields and the method invocation history of the object; JR [27], which extends JAVA, providing a rich concurrency model with a variety of mechanisms for writing parallel and distributed programs; and STATEJ [12], which proposes *state classes*, a construct for making the state of a concurrent object explicit. The objective of our approach is significantly more extensive in a sense, because we introduce an abstraction layer which aims at providing an effective support for tackling not only synchronization and coordination issues, but also the engineering of passive and active parts of the application, avoiding the direct use of low-level mechanisms such as threads.

The introduction of a higher level of concurrency is also found in JAC [21], which introduces only a few basic constructs that make it possible to avoid threads and separate thread synchronization from application logic in a declarative fashion. JAC actually adopts for Java the concept of concurrency annotations similar to the ones introduced by SCOOP: the aim is to minimize the differences between concurrent and sequential implementations of objects and computations. This is a main difference with respect to our approach, in which the new abstraction layer is meant to be on top and somewhat orthogonal to the object-oriented layer: classes and objects in simpA are used as the abstractions to define data structures (with no concurrent mechanisms), while agents and artifacts as abstractions to organize the concurrent program.

Patterns in concurrent programming — Doug Lea's patterns for concurrent programming in Java – which are a main reference also for the *java.util.concurrent* library – are another main related work [30,19]. Generally speaking, taken individually, the various features that characterize agents and artifacts in simpA can be related to some specific existing pattern. For instance: on the artifact side, the guards exploited in operation and operation steps are similar to the guarded method patterns; on the agent side, the definition and execution of sub-activities (tasks to be executed) in the context of a structured activity recall the pattern of parallel decomposition, used to structure services in threads.

Actually, complex concurrent programs require the composition and integration of different kinds of pattern. Each pattern provides an effective and efficient way to solve a specific (recurrent) concurrent programming problem, without raising the basic abstraction level provided by object-orientation and threads: however, the composition of multiple patterns to create a concurrent program is still a challenging issue, both because such integration is not as simple as the application of the individual pattern, and because there is no clear formal semantics of the patterns themselves.

An important example concerns the *Executor framework* (recalled in Section 3.4), available in the `java.util.concurrent` library. The framework provides direct support for breaking down a program into tasks and executing tasks according to execution policies encapsulated in proper objects that hide the management of threads. However, the approach can be used – generally speaking – only if the tasks are fully independent activities, i.e., each task does not depend on the state, result, or side effects of *other tasks* submitted to the same executor or other executors. This is because executors uncouple task execution from threads that execute tasks, so as to create an abstraction barrier between tasks and threads. Then, if we block the execution of a task – waiting, for instance, for the completion of another task – by exploiting basic synchronization mechanisms (such as semaphores, barriers, etc.), we are also going to block the thread which is executing the task. In that way we break the abstraction barrier: suppose in fact that the task can be unblocked only by the execution of another task submitted to the same executor, and that no more threads are available for task execution—because of the kind of policy chosen for the executor, e.g., fixed thread pool. Then we get a deadlock, even if – by reasoning at the task abstraction level – we should not. So, it is not immediately clear how to extend this approach in a general way or to integrate it with other patterns so as to exploit the task-oriented approach—keeping the decoupling between task submission and task execution but also managing in a flexible way possible dependencies among the tasks. This is precisely what structured activities inside simpA agents are useful for. Actually, the Fork–Join framework [31], which will be introduced with Java 7, allows for solving the problem for a specific – even if quite common – case, which is about running computationally intensive tasks that do not ever block except to wait – in the join stage – for another task being processed by the same executor.

So, the value of an approach like simpA in this case must be framed more in the direction devised in [50], namely, to introduce higher-level abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs", as recalled in Section 1. We believe that such high abstractions should, on the one hand, internally exploit as much as possible existing patterns, and, on the other hand, provide a programming model hiding as much as possible the complexity related to their combined use.

6. Conclusion and future works

More and more concurrency is going to be part of mainstream programming and software engineering, with applications able to suitably exploit the inherent concurrency support provided by modern hardware architectures, such as multi-core architectures, and by network-based environments and related technologies, such as Internet and Web services. This calls for tackling the problem at the right level of abstraction, and not through low-level mechanisms [50].

Along this line, in this paper we presented simpA, a framework on top of the Java platform that aims at simplifying the development of concurrent programs by introducing a high-level agent-oriented abstraction layer over the standard OO layer: alongside a description of the programming framework and its use, we also provided an operational semantics for a significant subset of the language, based on a calculus of agents and artifacts.

Future works will first of all be focussed on improving the effectiveness of the simpA approach. A main issue is related to the choice of realizing simpA as an annotation-based framework on top of the basic Java environment. Besides the mentioned advantage related to compatibility and ease of deployment, this choice is less flexible and effective than a solution relying on an entirely new language for programming agents and artifacts, affecting in particular programming language usability and program debugging — i.e., using strings in annotations prevents compile-time checking of correctness. This will make it possible to provide a deeper comparison with the reference programming languages adopted for building concurrent software — such as Erlang. Compared to the many existing general-purpose programming languages devised for concurrent programming, the objective of this language will be to investigate the use of the proposed abstractions for general-purpose programming in the context of concurrent/multi-core and distributed applications. Actually, many agent programming languages have already been described in the literature [9]: however, these languages are not meant to be general purpose and are typically targeted to the development of *intelligent* systems.

The definition of a language will be important also to ease the investigation of further aspects which are important from a computer programming and software engineering perspective: among others, the notion of *type* for the agent and artifact concept and related aspects concerning, for instance, a notion of inheritance and extension to support specialization and reuse for such abstractions. These issues have been already investigated, for instance, in the context of object-oriented concurrent languages, leading to the identification of well-known problems like the inheritance anomaly [33]. So, the language and related investigation will be useful also to understand if and how an agent-oriented abstraction layer makes it possible to frame these problems in a different way, possibly providing new kinds of solutions.

Another research direction concerns the analysis of properties of simpA programs. As mentioned, our plan for tackling this problem is to conceive a type system on top of the proposed calculus, by which it would be possible to intercept standard typing errors (e.g. accessing a non-defined field or operation), but also behavioral errors related to the possible computation paths (like reading memos that are surely not present, or declaring guards that cannot be positively evaluated). The goal of this research line is to ground the development of a fully featured compiler for the simpA approach.

Finally, an interesting work is to identify a small abstract core calculus based on agents and artifacts, which can be a suitable basis for systematically analysis the properties of systems of agents and artifacts. First investigations about this point can be found in [13], while future research will also be devoted to finding the suitable formal framework for specifying the language semantics, possibly relying on modular operational semantics [37], formalization by abstract machines as in [49], and barbed congruence [48].

References

[1] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.
[2] G. Agha, C.J. Callsen, ActorSpace: an open distributed programming paradigm, SIGPLAN Not. 28 (7) (1993) 23–32.
[3] G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-oriented Programming, MIT Press, Cambridge, MA, USA, 1993.
[4] F. Arbab, The IWIM model for coordination of concurrent activities, in: COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models, Springer-Verlag, London, UK, 1996, pp. 34–56.
[5] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[6] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, Decidability of bisimulation equivalence for processes generating context-free languages, in: PARLE, Parallel Architectures and Languages Europe, Volume I: Parallel Languages, in: Lecture Notes in Computer Science, vol. 259, Springer, 1987, pp. 94–111.
[7] F.L. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE, Wiley, 2007.
[8] N. Benton, L. Cardelli, C. Fournet, Modern concurrency abstractions for C#, ACM Trans. Program. Lang. Syst. 26 (5) (2004) 769–804.
[9] R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), Multi-Agent Programming Languages, Platforms and Applications, vol. 15, Springer, 2005.
[10] J.-P. Briot, R. Guerraoui, K.-P. Lohr, Concurrency and distribution in object-oriented programming, ACM Comput. Surv. 30 (3) (1998) 291–329.
[11] G. Caire, D. Gotta, M. Banzi, WADE: a software platform to develop mission critical applications exploiting agents and workflows, in: AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008, pp. 29–36.
[12] F. Damiani, E. Giachino, P. Giannini, S. Drossopoulou, A type safe state abstraction for coordination in Java-like languages, Acta Inf. 45 (7–8) (2008) 479–536.
[13] F. Damiani, P. Giannini, A. Ricci, M. Viroli, Featherweight agent language — a core calculus for agents and artifacts, in: B. Shishkov, J. Cordeiro, A. Ranchordas (Eds.), ICSOFT 2009 — Proceedings of the 4th International Conference on Software and Data Technologies, Volume 1, Sofia, Bulgaria, July 26–29, 2009, INSTICC Press, 2009, pp. 218–225.
[14] E. Denti, A. Natali, A. Omicini, Programmable coordination media, in: D. Garlan, D. Le Métayer (Eds.), Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION'97), Berlin (D), 1–3 Sept. 1997, in: LNCS, vol. 1282, Springer-Verlag, 1997, pp. 274–288.
[15] C. Dufourd, A. Finkel, P. Schnoebelen, Reset nets between decidability and undecidability, in: Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13–17, 1998, Proceedings, in: Lecture Notes in Computer Science, vol. 1443, Springer, 1998, pp. 103–115.
[16] C. Fournet, G. Gonthier, The reflexive chemical abstract machine and the join calculus, in: POPL'96, ACM, 1996, pp. 372–385.
[17] S. Frolund, G. Agha, Abstracting interactions based on message sets, in: ECOOP '94: Selected Papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems, Springer-Verlag, London, UK, 1995, pp. 107–124.
[18] D. Gelernter, Generative communication in Linda, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 80–112.
[19] B. Goetz, et al., Java Concurrency in Practice, Addison-Wesley, 2006.
[20] P. Haller, M. Odersky, Scala actors: unifying thread-based and event-based programming, Theoret. Comput. Sci. (2008).
[21] M. Haustein, K.-P. Loehr, JAC: declarative Java concurrency: research articles, Concurr. Comput.: Pract. Exper. 18 (5) (2006) 519–546.
[22] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Trans. Program. Lang. Syst. 23 (2001) 396–450.

[23] A. Igarashi, M. Viroli, Variant parametric types: a flexible subtyping scheme for generics, ACM Trans. Program. Lang. Syst. 28 (5) (2006) 795–847.
[24] C. Iglesias, M. Garrijo, J. Gonzalez, A survey of agent-oriented methodologies, in: J. Müller, M.P. Singh, A.S. Rao (Eds.), in: Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages, ATAL-98, vol. 1555, Springer-Verlag, Heidelberg, Germany, 1999, pp. 317–330.
[25] G.S. Itzstein, D. Kearney, Join Java: an alternative concurrency semantics for Java, Technical Report ACRC-01-001, University of South Australia, 2001.
[26] N.R. Jennings, An agent-based approach for building complex software systems, Commun. ACM 44 (4) (2001) 35–41.
[27] A.W. Keen, T. Ge, J.T. Maris, R.A. Olsson, JR: flexible distributed programming in an extended Java, TOPLAS 26 (3) (2004) 578–608.
[28] J. Larson, Erlang for concurrent programming, Commun. ACM 52 (3) (2009) 48–56.
[29] R.G. Lavender, D.C. Schmidt, Active object: an object behavioral pattern for concurrent programming, in: Pattern Languages of Program Design 2, Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, USA, 1996, pp. 483–499.
[30] D. Lea, Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley, 1999.
[31] D. Lea, A Java Fork/Join framework, in: JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande, ACM, New York, NY, USA, 2000, pp. 36–43.
[32] D. Lea, The java.util.concurrent synchronizer framework, Sci. Comput. Programming 58 (3) (2005) 293–309.
[33] S. Matsuoka, A. Yonezawa, Analysis of inheritance anomaly in object-oriented concurrent programming languages, in: G. Agha, A. Yonezawa, P. Wegner (Eds.), Research Directions in Concurrent Object-Oriented Programming, The MIT Press, 1993.
[34] B. Meyer, Systematic concurrent object-oriented programming, Commun. ACM 36 (9) (1993) 56–80.
[35] G. Milicia, V. Sassone, Jeeg: temporal constraints for the synchronization of concurrent objects, Concurr. Comput.: Pract. Exper. 17 (5–6) (2005) 539–572.
[36] R. Milner, Communicating and Mobile Systems: The π-calculus, Cambridge University Press, 1999.
[37] P.D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60–61 (2004) 195–228.
[38] A. Omicini, E. Denti, From tuple spaces to tuple centres, Sci. Comput. Programming 41 (3) (2001) 277–294.
[39] A. Omicini, A. Ricci, M. Viroli, Artifacts in the A&A meta-model for multi-agent systems, Auton. Agents and Multi-Agent Syst. 19 (2009) Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.
[40] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, L. Tummolini, Coordination artifacts: environment-based coordination for intelligent agents, in: AAMAS'04, vol. 1, ACM, New York, USA, 19–23 July, 2004, pp. 286–293.
[41] A. Pokahr, L. Braubach, W. Lamersdorf, Jadex: a BDI reasoning engine, in: R. Bordini, M. Dastani, J. Dix, A. Seghrouchni (Eds.), Multi-Agent Programming, Kluwer, 2005.
[42] M. Resnick, Turtles, Termites and Traffic Jams. Explorations in Massively Parallel Microworlds, MIT Press, 1994.
[43] A. Ricci, M. Piunti, M. Viroli, A. Omicini, Environment programming in CArtAgO, in: R.H. Bordini, M. Dastani, J. Dix, A. El Fallah-Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, vol. 2, Springer, 2009, pp. 259–288.
[44] A. Ricci, M. Viroli, simpA: An agent-oriented approach for prototyping concurrent applications on top of Java, in: V. Amaral, L. Veiga, L. Marcelino, H.C. Cunningham (Eds.), 5th International Conference, Principles and Practice of Programming in Java, PPPJ 2007, Lisbon, Portugal, 5–7 September 2007, pp. 185–194.
[45] A. Ricci, M. Viroli, M. Cimadamore, Prototyping concurrent systems with agents and artifacts: framework and core calculus, in: 6th International Workshop on Foundations of Coordination Languages and Software Architectures, FOCLASA'07, CONCUR'07, Lisbon, Portugal, 8 September 2007. Proceedings, Electron. Notes Theor. Comput. Sci. 194 (4) (2008) 111–132.
[46] A. Ricci, M. Viroli, A. Omicini, Give agents their artifacts: the A&A approach for engineering working environments in MAS, in: E. Durfee, M. Yokoo, M. Huhns, O. Shehory (Eds.), 6th International Joint Conference Autonomous Agents & Multi-Agent Systems, AAMAS 2007, IFAAMAS, Honolulu, Hawai'i, USA, 14–18 May 2007, 2007, pp. 601–603.
[47] A. Ricci, M. Viroli, G. Piancastelli, simpA: a simple agent-oriented Java extension for developing concurrent applications, in: Languages, Methodologies and Development Tools for Multi-Agent Systems, in: LNCS, vol. 5118/2008, Springer, Berlin, Heidelberg, July 2008, pp. 261–278, First International Workshop, LADS 2007, Durham, UK, September 4–6, 2007, Revised Selected Papers.
[48] D. Sangiorgi, D. Walker, On barbed equivalences in pi-calculus, in: CONCUR 2001 — Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20–25, 2001, Proceedings, in: Lecture Notes in Computer Science, vol. 2154, Springer, 2001, pp. 292–304.
[49] P. Sewell, On implementations and semantics of a concurrent programming language, in: CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1–4, 1997, Proceedings, in: Lecture Notes in Computer Science, vol. 1243, Springer, 1997, pp. 391–405.
[50] H. Sutter, J. Larus, Software and the concurrency revolution, ACM Queue: Tomorrow's Comput. Today 3 (7) (2005) 54–62.
[51] H. Velroyen, P. Rümmer, Non-termination checking for imperative programs, in: Tests and Proofs, in: Lecture Notes in Computer Science, vol. 4966/2008, Springer, 2008, pp. 154–170.
[52] M. Viroli, A core calculus for correlation in orchestration languages, J. Log. Algebr. Program. 70 (1) (2007) 74–95, Special Issue on Web Services and Formal Methods.
[53] M. Viroli, E. Denti, A. Ricci, Engineering a BPEL orchestration engine as a multi-agent system, Sci. Comput. Programming 66 (3) (2007) 226–245.
[54] D. Weyns, H.V.D. Parunak (Eds.), J. Auton. Agents Multi-Agent Syst. 14 (1) (2007) Special Issue: Environment for Multi-Agent Systems.
[55] A.K. Wright, M. Felleisen, A syntactic approach to type soundness, Inform. Comput. 115 (1) (1994) 38–94.
[56] A. Yonezawa, M. Tokoro (Eds.), Object-oriented Concurrent Programming, MIT Press, Cambridge, MA, USA, 1986.