Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 194 (2008) 111–132
www.elsevier.com/locate/entcs

Prototyping Concurrent Systems with Agents and Artifacts: Framework and Core Calculus

Alessandro Ricci, Mirko Viroli, Maurizio Cimadamore

DEIS, Alma Mater Studiorum, Università di Bologna,
via Venezia 52, 47023 Cesena, Italy
{a.ricci,mirko.viroli,maurizio.cimadamore}@unibo.it

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. Besides the typical fine-grained support currently provided, however, we seek in this paper for an higher-level approach. We present simpA, a library-based extension of Java which provides programmers with *agent* and *artifact* abstractions on top of the basic OO layer, as a means to organise and structure concurrent applications. To pave the way towards identifying a true language extension for simpA, we define a core calculus of agents and artifacts, by suitabling mixing techniques coming from object-orientation and concurrency theory.

Keywords: multiprogramming, agents and artifacts, Multiagent systems, concurrent programming, core calculi

1 Introduction

The widespread diffusion and availability of parallel machines given by multicore 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, scenarios like Internet-based computing and Service-Oriented Architectures / Web Services are introducing concurrency issues in the context of a large class of applications and systems, not just to specific and narrow domains—such as high-performance scientific computing. As noted in [19], 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 significantly impacted on mainstream software development.

Considering as a reference the Java programming framework, we note that Java has been one of the first mainstream languages providing a first-class native support for multi-threading, with basic low level concurrency mechanisms. This has been recently extended with a new library implementing well-known and useful higher-

doi:10.1016/j.entcs.2008.03.102

level synchronisation mechanisms such as barriers, latches and semaphores, which provide a *fine-grained* and efficient control on concurrent computations [10]. However, it appears more and more important to also introduce higher-level abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [19].

Accordingly, in this paper we present simpA, a library-based extension of Java which provides programmers with agent-oriented abstractions on top of the basic OO layer, as basic building blocks to define the architecture of complex (concurrent) applications. simpA is based on the A&A (Agents and Artifacts) meta-model, recently introduced in the context of agent-oriented computing as a novel foundational approach for engineering complex software systems [18,14]. *Agents* and *artifacts* are the basic high-level, coarse-grained abstractions available in A&A (and simpA): agents are used in A&A to model (pro)-active and task-oriented components of a system, encapsulating the logic and control of such activities, while artifacts are used to model function-oriented components of the system, used by agents to support their individual activities, as well as collective ones.

Then, towards identifying a true Java language extension supporting the simpA framework, in this paper we identify a quite expressive subset of simpA, and accordingly develop a core calculus of agents and artifacts. To properly join the object-oriented nature of mainstream programming languages with the concurrency aspects of agents and artifacts, this formal framework suitably mixes modelling techniques coming from object-orientation [7,22] and concurrency theory [3,12]. As a result, this calculus paves the way toward abstract analysis of properties concerning well-formedness (typing), safety and liveness.

The remainder of the paper is organised as follows. Section 2 describes in more details the basic abstraction layer introduced by the A&A meta-model; Section 3 describes the simpA framework and technology; Section 4 describes a core calculus providing a formal foundation for the approach. Finally, Section 5 and Section 6 conclude the paper with related works and a brief sum up.

2 Agents and Artifacts

As recently remarked by Liebermann [11]:

> "The history of Object-Oriented Programming can be interpreted as a continuing quest to capture the notion of *abstraction*—to create computational artifacts that represent the essential nature of a situation, and to ignore irrelevant details."

Metaphors and abstractions continue to play a fundamental role for computer science and software engineering: they provide suitable conceptual means to model, design and program software systems. The metaphors and abstractions at the core of A&A are rooted in human cooperative working environments, investigated by socio-psychological theory such as Activity Theory (AT) [13]. This context is taken here as a reference example of a complex system, which features multiple concurrent activities carried on in a coordinated manner, interacting within some kind of

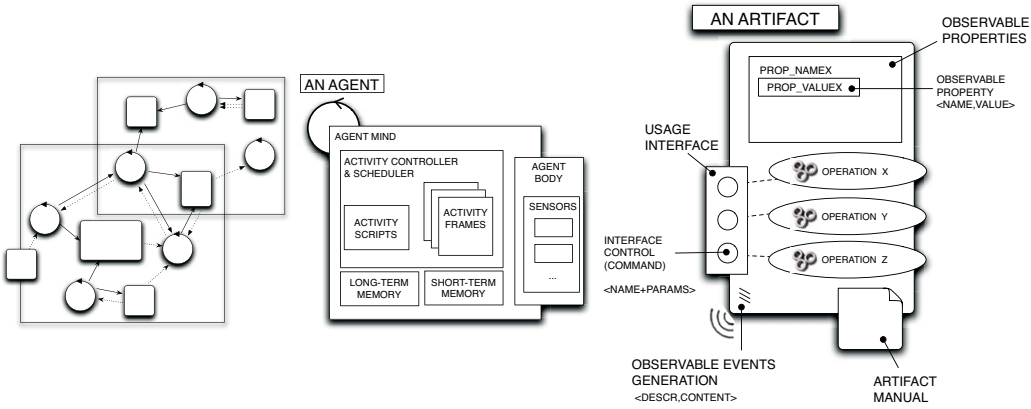

Fig. 1. *(Left)* An abstract representation of an application according to the A&A programming model, as a collection of agents (circles) sharing and using artifacts (squares), grouped in workspaces. *(Center)* An abstract representation of an agent, as an entity executing actions and getting perceptions from the environment where it is logically situated. *(Right)* An abstract representation of an artifact, as an entity providing a usage interface of operations, and generating observable events.

working environment: humans work concurrently and cooperatively in the context of social activities, directly interacting by means of speech-based communication, and indirectly by means of artifacts and tools that are shared and co-used. In such systems, it is possible to easily identify two basic kinds of entity: on the one side human workers, as the entities responsible of pro-actively performing some kinds of activity; on the other side artifacts and tools, as the entities that workers use to support their activities, being resources (e.g. an archive) or instruments mediating and coordinating collective activities (e.g. a blackboard, a calendars, a task schedulers).

By drawning our inspiration from AT and human working environments, A&A defines a coarse-graned abstraction layer in which two basic building blocks are provided to organise an application (system), *agents* and *artifacts*. On the one hand, the agent abstraction—in analogy with human workers—is meant to model the active / task-oriented part of the system, encapsulating the logic and the control of such activities. On the other hand, the artifact abstraction—analogous to artifacts and tools in human environments—is meant to model the resources and the tools created and used by agents during their activities. Then, A&A provides a meta-model for software architecture where agents and artifacts are the basic types of components and connectors: agents can be seen as active components, artifacts can be seen as both passive components and connectors.

Besides agents and artifacts, the notion of *workspace* completes the basic set of abstractions defined in A&A: a workspace is a logic container of artifacts, and it is the basic mean to give an explicit (logical) topology to the working environment, providing a way to scope the interaction inside it (see Fig. 1, left).

2.1 Agent and Artifact Abstractions: Core Properties

In A&A the term "agent" is used in its etymological meaning of an entity "who acts", i.e., whose computational behaviour accounts for performing *actions* in some kind

of environment and getting information back in terms of *perceptions* (see Fig. 1, center) [6]. In A&A agent actions and perceptions will concern the use of artifacts. The notion of *activity* is used to group related actions, as a way to structure the overall (possibly complex) behaviour of the agent. So an agent in A&A is an *activity-oriented* component, in the sense that it is designed to encapsulate the logic, execution and control of some activities, targeted to the achievement of some objective. As a stateful entity, each agent has a *long-term memory*, used to store data and information needed for its overall work, and a *short-term memory*, used as a working memory to store temporary information useful within single activities. An agent can carry on multiple activities concurrently, and each activity in execution defines a context, a local scope for storing information related to the specific activity, for executing actions and perceiving events from the agent environment. It is worth remarking here that differently from components as typically found in software architectures, agents have no interface: their interaction with the environment takes place solely in terms of actions and perceptions.

Artifacts instead are passive *function-oriented* components, i.e., designed to provide some kind of functionality that can be used by agents. Passive here means that, differently from the agent case, they do not encapsulate any thread of control. The functionality of an artifact is structured in terms of *operations*, whose execution can be triggered by agents through a *usage interface* (see Fig. 2, left). Similarly to the notion of interface in case of objects or components, the usage interface of an artifact defines a set of commands (interface controls) that an agent can trigger so as to execute operations, each one identified by a label (typically equals to the operation name to be triggered) and a list of input parameters. Differently from the notion of interface as found in OO, here there is no control coupling: when an agent triggers the execution of an operation, it retains its control (flow) and the execution of the operation on the artifact is carried on independently and asynchronously. This property is a requirement when the basic notion of agent autonomy is considered.

The information flow from the artifact to agents is modelled in terms of *observable events* generated by artifacts and perceived by agents (so interface controls have no return values, as found in OO interfaces). Besides the controls for triggering the execution of operations, an artifact can have some *observable properties*, i.e., elements useful to inspect the dynamic state of the artifact without necessarily executing operations on it.

2.2 Agent-Artifact Interaction: Use and Observation

The interaction between agents and artifacts strictly mimics the way in which humans use their artifacts: let's consider a coffee machine, for a simple but effective analogy. The set of buttons of the coffee machines represents the usage interface, while the displays that are typically used to show the state of the machine represent artifact observable properties. The signals emitted by the coffee machine during its usage represent observable events generated by the artifact.

The interaction takes place by means of a *use* action (stage 1a in Fig. 2, center), which is provided to agents so as to trigger the execution of an operation over

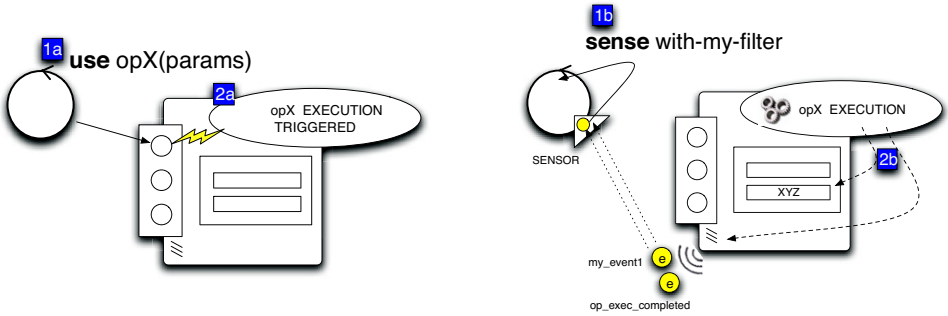

Fig. 2. Abstract representation of artifact (on the left), and of an agent using an artifact, by triggering the execution of on operation (center, step 1a) and observing the related events generated (right, step 1b)

an artifact by specifying the name and parameters of the interface control (which corresponds to the name and parameters of the operation triggered by the control). The observable events possibly generated by the artifact executing an operation are collected by agent *sensors*, which are the parts of the agent (body) conceptually connected to the environment where the agent is situated. Besides the generation of observable events, the execution of an operation by an artifact typically results in updating the artifact inner state and possibly artifact observable properties (Fig. 2, right).

Then, a *sense* action is provided to agents so as to explicitly retrieve (or be aware of) the observable events possibly collected by their sensors (stage 1b in Fig. 2, right); in other words, there is an "active sensing" model for managing perceptions, since sensing—i.e. making the agent aware of the stimuli collected by the sensors—is an action that must be explicitly performed by the agent itself.

As mentioned previously no control coupling takes place between an agent and an artifact during the execution of an operation. However, the triggering of an operation is a synchronisation point between the agent (user) and the artifact (used): if the use action is successfully executed, then it means that the execution of the operation on the artifact has started.

3 The simpA Framework & Technology

simpA is an extension of the Java platform that supports the A&A abstractions as first-class concepts, treating them as basic high-level building blocks to program concurrent applications. This approach contrasts most existing ones that modify object-oriented abstractions (classes, objects, methods) to model concurrency aspects—such as e.g. [4]. Rather, we introduce the new A&A abstractions, and use true object-orientation to model any basic low-level data structure used to program agents and artifacts, or any information kept and exchanged by them through interactions. This approach leaves concurrency and high-level organisation aspects orthogonal to the object-oriented abstraction layer: we argue this approach could lead to a more coherent programming framework for complex applications.

simpA extension is realised as a library, exploiting Java annotations to define the new programming constructs required: consequently, a simpA program can be com-

piled and executed using the standard Java compiler and virtual machine, without the need of a specific extension of the Java framework (preprocessors, compilers, class loaders, or JVM patches). This choice has the advantage to maximise the reuse of an existing widely diffused platform (Java). Indeed, using the library / annotations solution to implement a language and a platform extension has some revelant drawbacks, which derive from the fact that agents and artifacts are not true real first-class abstractions for the language and the virtual machine. Accordingly, part of the ongoing work is devoted towards the definition and the prototype implementation of a new full-fledged language and platform called simpAL.

In the remainder of the section we give a more concrete taste of the A&A approach by describing how an application based on agents and artifacts can be designed and programmed on top of simpA. Table 1 reports the source code of a simple application used as a reference to exemplify the definition of an agent and artifact in simpA. This example is part the basic examples provided in simpA distribution, available on simpA web site.

The application creates a simple `Cafeteria` workspace, composed of a single `Waiter` agent using two instances of a `CoffeeMachine` artifact. The `CoffeeMachine` artifact mimics the behaviour of a coffee machine: it can be used to make either coffees or teas. Essentially, it provides a usage interface with controls for first selecting the type of drink (coffee or tea), then to make the drink. Then, while making the drink, it provides a usage interface to adjust the sugar level and possibly to stop the operation (for a short drink). The `Waiter` agent is programmed with the objective to make a coffee and a tea by exploiting two different coffee machines, and to deliver them when both are ready within a certain amount of time, or just the coffee if the tea making lasts too much. A simpA application typically sets up the workspace(s), creating an initial sets of artifacts—two `CoffeeMachine`s in the example—and spawning agents—a single `Waiter` in this case. For this purpose, the `Simpa` class and the `ISimpaEnvironment` interface provide suitable services to initialise and configure the working environment.

3.1 Defining Agents

A requirement in simpA was to make the approach as agile as possible, minimising the number of classes to be introduced for defining both agents and artifacts. For that reason a one-to-one mapping has been adopted: just one class is needed to define an agent template or an artifact template. Accordingly, to define a new agent (template), only one class must be defined, extending the `alice.simpa.Agent` base class provided by simpA API. The class name corresponds to the agent template name. The elements defining an agent are mapped into suitably-annotated class elements. By defining a template, it is possible at runtime to spawn an instance of such type of agent. The execution of an agent consists in executing the activities as specified in its template, starting from the `main` one.

Agent long-term memory is realised as an associative store called *memo-space*, where the agent can dynamically attach, associatively read and retrieve chunks of information called *memo*. A memo is a tuple, characterised by a label and an

ordered set of arguments, either bound or not to some data object (if some is not bound, the memo is hence partially specified). A memo-space is just a dynamic set of memos: a memo is identified by its label, and only one instance of a memo can exist at a time. Each agent has internal actions to atomically and associatively access and manipulate the memo space: to create a new memo, to get/remove a memo with the specified label and/or content, and so on. It is worth remarking here that instance fields of an agent class are not used: the memo-space is the only data structure adopted for modelling agent long-term memory.

Agent activities can be either *atomic*—i.e., not composed by sub-activities—or *structured*, composed by some kinds of sub-activity. Atomic activities are implemented as methods with the `@ACTIVITY` annotation, with no input parameters and with `void` return type. The body of a the method specifies the computational behavior of the agent corresponding to the accomplishment of the activity. Method local variables are used to encode data-structures representing the short-term memory related to the specific activity. By default, the main activity of an agent is called `main`, and must be defined by every new agent tamplate. By referring to the example reported in Table 1, a `Waiter` agent has four atomic activities: `makeOneCoffee`, `makeOneTea`, `deliverBoth`, `deliverJustCoffee`. The activity `main` is not atomic, but structured.

Structured activities can be described as activities composed (hierarchically) by sub-activities. The notion of *agenda* is introduced to specify the set of the potential sub-activities composing the activity, referenced as *todo* in the agenda. Each todo specifies the name of the subactivity to execute, and optionally a pre-condition. When a structured activity is executed, each todo in the agenda is executed as soon as its pre-condition holds. If no pre-condition is specified, the todo is immediately executed. Then, multiple sub-activities can be executed concurrently in the context of the same (super) activity. A structured activity is implemented by methods with an `@ACTIVITY_WITH_AGENDA` annotation, containing todo descriptions as a list of `@TODO` annotations. Each `@TODO` must specify the name of the related sub-activity to execute and optionally a `pre` property specifying the precondition that must hold in order to execute the todo. A todo can be specified to be *persistent*: in that case, once it has been completely executed, it is re-inserted in the agenda so as to be possibly executed again. This is useful to model cyclic behaviours. Todo preconditions are expressed as boolean expressions, with and / or connectors (represented by `,` and `;` symbols, respectively) over a basic set of predefined predicates. Essentially, such predicates make it possible to specify conditions on the current state of the activity agenda, in particular on *(i)* the state of the sub-activitities (todo), if they completed or aborted or started, and on *(ii)* the memos that could have been attached to the agenda. Besides holding information useful for activities, memos are used then also to coordinate activities, by exploiting in the specification of a pre-condition the predicate `memo`, which tests the presence of a memo in the agenda.

By referring to the example reported in Table 1, the `Waiter` has a structured `main` activity, with four todos: making a coffee (`makeOneCoffee`) and making a tea (`makeOneTea`), as activities that can be performed concurrently as soon as the main

activity starts, and then either delivering the drinks (`deliverBoth`) as soon as both the drinks are ready, or deliver just the coffee (`deliverJustCoffee`) if only the tea is not available after a specific amount of time. At the end of the activities, the primitive `memo` is used to create memos about the drinks (labelled with `drink1` and `drink2`), annotating information related to the fact that coffee and the tea are done. In the case of `makeOneTea` activity, the memo `tea_not_ready` is created instead if the agent does not perceive that the tea is ready within a specific amount of time. In `deliverJustCoffee` and `deliverBoth` activities the primitive `getMemo` is used instead to retrieve the content of a memo.

To perform their activities, agents typically need to interact with their working environment, in particular with artifacts by means of *use* and *sense* actions as described in previous section. For this purpose, the `use` and `sense` primitives are provided, respectively, to trigger the execution of an operation over an artifact, and for perceiving the observable events generated by the artifact as effect of the execution. Before describing in details agent-artifact interaction, in next sub-section we describe how to programs artifacts.

3.2 Defining Artifacts

Analogously to agents, also each artifact is mapped onto a single class. An artifact template can be described by a single class extending the `alice.simpa.Artifact` base class. The elements defining an artifact—its inner and observable state and the operations defining its computational behaviour—are mapped into suitably-annotated class elements. The instance fields of the class are used to encode the inner state of the artifacts and observable properties, while suitably-annotated methods are used to implement artifact operations.

For each operation (command) listed in the usage interface, a method annotated with `@OPERATION` and with `void` return type must be defined: the name and parameters of the method coincide with the name and parameters of the operations to be triggered. Operations can be either *atomic*, i.e. executed as a single computational *step* represented by the content the `@OPERATION` method, or *structured*, i.e. composed by multiple atomic steps. Structured operations are useful to implement those services that would need multiple interactions with—possibly different—agents (users of the artifact), and that cannot be provided "in one shot". A structured operation is implemented by dynamically specifying the operation steps composing the operation. 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 to be enabled and possibly its parameters.

For each operation and operation step a *guard* can be specified, i.e. a condition that must be true in order to actually execute the operation / step after it has been enabled (triggered). Guards are implemented as boolean methods annotated with the `@GUARD` annotation, with same parameters as the guarded operation (step). The step is actually executed as soon as its guard is evaluated to true. Guards can be specified also for an operation, directly. Also *temporal* guards are supported, i.e. guards whose evaluation is true when a specific delta time is elapsed after triggering.

```
public class Waiter extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="makeOneCoffee"),
    @TODO(activity="makeOneTea"),
    @TODO(activity="deliverBoth",
          pre="completed(makeOneCoffee),
               completed(makeOneTea)"),
    @TODO(activity="deliverJustCoffee",
          pre="completed(makeOneCoffee),
               memo(tea_not_ready)"),
  }) void main(){}

  @ACTIVITY void makeOneCoffee() {
    SensorId sid = linkDefaultSensor();
    ArtifactId id = lookupArtifact("cmOne");

    use(id, new Op("selectCoffee"));
    use(id, new Op("make"),sid);
    sense(sid,"making_coffee");

    focus(id,sid);
    Perception p = null;
    do {
      use(id, new Op("addSugar"));
      p = sense(sid,
          "property_updated\\(\"sugarLevel\"\\)");
    } while ((Double)(p.getContent())<0.5);

    Perception p1 = sense(sid,"coffee_ready",5000);
    memo("drink1",coffep.getContent(0));
  }

  @ACTIVITY void makeOneTea() {
    SensorId sid = linkDefaultSensor();
    ArtifactId id = lookupArtifact("cmTwo");

    use(id, new Op("selectTea"));
    use(id, new Op("make"),sid);
    try {
      Perception p = sense(sid,"tea_ready",6000);
      memo("drink2",coffep.getContent(0));
    } catch (NoPerceptionException ex){
      memo("tea_not_ready");
    }
  }

  @ACTIVITY void deliverBoth() {
    log("delivering "+
        getMemo("drink1").getContent(0)+" "+
        getMemo("drink2").getContent(0));
  }

  @ACTIVITY void deliverJustCoffee() {
    log("delivering only "+
        getMemo("drink1").getContent(0));
  }
}

public class Testcafeteria {
  public static void main(String[] args){
    ISimpaEnvironment env =
                Simpa.getInstance("Cafeteria");
    env.createArtifact("cmOne","CoffeeMachine");
    env.createArtifact("cmTwo","CoffeeMachine");
    env.spawnAgent("waiter","Waiter");
  }
}
```

```
@ARTIFACT_MANUAL(
  states = {"idle","making"},
  start_state = "idle" )
class CoffeeMachine extends Artifact {

  @OBSPROPERTY String selection = "";
  @OBSPROPERTY double sugarLevel = 0.0;

  int nCupDone=0;
  boolean makingStopped;

  @OPERATION(states={"idle"})
  void selectCoffee(){
    updateProperty("selection", "coffee");
  }

  @OPERATION(states={"idle"})
  void selectTea(){
    updateProperty("selection", "tea");
  }

  @OPERATION(states={"idle"})
  void make(){
    if (selection.equals("")){
      signal("no_drink_selected");
    } else {
      makingStopped = false;
      switchToState("making");
      signal("making_"+selection);
      nextStep("timeToReleaseDrink");
      nextStep("forcedToReleaseDrink");
    }
  }

  @OPSTEP(tguard=3000)
  void timeToReleaseDrink(){
    releaseDrink();
  }

  @OPSTEP(guard="makingStopped")
  void forcedToReleaseDrink(){
    releaseDrink();
  }

  private void releaseDrink(){
    String drink = selection+
                   "("+(++nCupDone)+
                   ","+sugarLevel+")";
    signal(selection+"_ready",drink);
    updateProperty("selection", "");
    updateProperty("sugarLevel", 0);
    switchToState("idle");
  }

  @GUARD boolean makingStopped(){
    return makingStopped;
  }

  @OPERATION(states={"making"})
  void addSugar(){
    double sl = sugarLevel + 0.1;
    if (sl>1){ sl=1; }
    updateProperty("sugarLevel", sl);
  }

  @OPERATION(states={"making"})
  void stop(){
    makingStopped = true;
  }
}
```

Table 1
An example case of simpA application, composed at runtime by a single `Waiter` agent using two instances—`cmOne` and `cmTwo`—of the `CoffeeMachine` artifact

To define a temporal guard, a `tguard` property must be specified inside the `@OPSTEP` annotation in the place of `guard`: the property can be assigned with a long value greater that 0, indicating the number of milliseconds that elapse between triggering and actual execution.

Multiple steps can be triggered as next steps of an operation at a time: As soon as the guard of a triggered step is evaluated to true, the step is executed—in mutual exclusion with respect to the steps of the other operations in execution—and the other triggered steps of the operation are discarded. In other words, an operation execution is composed of a linear sequence of steps. If multiple steps are evaluated to be runnable at a time, one is chosen according to the order in which they have been triggered with the `nextStep` primitive. It is worth remarking that, in the overall, multiple structured operations can be in execution on the same artifact at the same time, but with only one operation step in execution at a time, enforcing mutual exclusion in accessing the artifact state.

To be useful, an artifact should typically provide some level of *observability*. This is achieved either by generating observable events through the `signal` primitive or by defining observable properties. In the former case, the primitive generates observable events that can be observed by the agent using the artifact—i.e. by the agent which has executed the operation. An observable event is represented by a tuple, with a label (string) representing the kind of the event, and a set of arguments, useful to specify some information content. In the latter case, observable properties are implemented as instance fields annotated with the `OBSPROPERTY` annotation. Any change of the property by means of the `updateProperty` primitive would generate an observable event of the type `property_updated(`*`PropertyName`*`)` with the new value of the property as a content. The observable event is observed by all the agents that are *focussing* on the artifact (observation). More on this will be provided in next subsection, when describing agent-artifact interaction.

Finally, the usage interface of an artifact can be partitioned in labelled states, in order to allow a different usage interface according to the specific functioning state of the artifact. This is realised by specifying the annotation property `states` when defining operations and observable properties: it describes the list of observable states in which the specific property / operation is visible. The primitive `switchToState` is provided to change the state of the artifact (changing then the exposed usage interface).

In the example reported in Table 1, the `CoffeeMachine` artifact has two basic functioning states, `idle` and `making`, with the former used as starting state. In the `idle` state, the usage interface is composed by `selectCoffee`, `selectTea` and `make` operations, the first two used to select the drink type and the third one to start making the selected drink; in the `making` state, the usage interface is composed by `addSugar` and `stop` operations, the first used to adjust the sugar level during drink-making and the last to stop the process so as to get shorter drinks. Also, it has two observable properties, `selection` which reports the type of the drink currently selected, and `sugarLevel` which reports the current level of sugar: when, for example, `selection` is updated by `updateProperty`, an observable event

`property_updated("selection")` is generated. The operations `selectCoffee` and `selectTea` are atomic, instead `make` is (can be) structured: if a valid drink selection is available, then two possible alternative operation steps are scheduled, `timeToReleaseDrink` and `forcedToReleaseDrink`. The first one is time-triggered, and is executed 3 seconds after triggering. The second one is executed as soon as `makingStopped` guard is evaluated to true. This can happen if the user agent executed the `stop` operation while the coffee machine was making the coffee. In both cases, step execution accounts for releasing the drink, by signaling a proper event of type `coffee_ready` or `tea_ready`, updating the observable properties value and switching to the `idle` state.

3.3 Agent-Artifact Interaction

Artifact *use* is the basic form of interaction between agents and artifacts. Actually, also artifact instantiation and artifact discovery are realised by means of using proper artifacts—a *factory* and a *registry* artifacts—which are available in each workspace. However, two high-level macros are provided, `makeArtifact` and `lookupArtifact`, which encapsulate the interaction with such artifacts.

Following the A&A model, artifact use by a user agent involves two basic aspects: (1) executing operations on the artifact, and (2) perceiving the observable events generated by the artifact through agent sensors.

Agents execute operations on an artifact by using the interface controls provided by the artifact usage interface. The `use` basic action is provided for this purpose, specifying the identifier of the target artifact, the operation to be triggered and optionally the identifier of the sensor used to collect observable events generated by the artifact. When the action execution succeeded, the use operation returns the operation unique identifier. If the action execution fails—for instance, because the interface control specified is not part of the artifact usage interface—, an exception is generated. An agent can link (and unlink) any number of sensors (of different kinds), according to the strategy chosen for sensing and observing the environment, by means of specific primitives (`linkSensor`, `unlinkSensor`, and `linkDefaultSensor`, to link a new default type of sensor).

In order to retrieve events collected by a sensor, the `sense` primitive is provided. The primitive waits until either an event is collected by the sensor, matching the pattern optionally specified as a parameter (for data-driven sensing), or when a timeout is reached. As a result of the successful execution of a `sense`, the event is removed from the sensor and a perception related to that event is return—represented by an object instance of the class `Perception`. If no perception is sensed during the time specified, the action generates a `NoPerceptionAvailableException`. Pattern-matching is based on regular-expression patterns, matched over the event type (a string).

Finally, a support for *continuous observation* is provided. If an agent is interested to observe every event generated by an artifacts—including those generated as a result of the interaction with other agents—two primitives can be used, `focus` and `unfocus`. The former is used to start observing the artifact, specifying a sensor

to be used to collect the events—and optionally the reg-ex filter to define the set of events to observe. The latter one is used to stop observing the artifact.

In the example reported in Table 1, in the `makeCoffee` activity the agent uses the coffee machine `cmOne` discovered by the `lookupArtifact` action, by executing first a `selectCoffee` operation—ignoring possible events generated by such operation execution—and then a `make`, specifying a sensor to collect events. Then the agent, by means of a `sense`, waits to observe a `making_coffee` event, meaning that the artifact started making the coffee. The agent then interacts with the machine so as to adjust the sugar level: this is done by focussing on the artifact and acting upon the `addSugar`, until the observable property reporting the sugar level reaches 0.5. Then the activity is blocked until `coffee_ready` event is perceived. While performing a `makeOneCoffee` activity, the agent carries on also a `makeOneTea` activity: as a main difference there, if the agent does not observe the `tea_ready` event within six seconds after triggering the `make` operation, then a memo `tea_not_ready` is taken and the activity fails (by means of the generation of an exception).

4 A core calculus

In this section we start studying a language supporting the main abstractions of the simpA framework, considering as a reference a simpA subset expressive enough to deal with powerful coordination artifacts. This calculus is conceived as a smooth extension over the object-oriented setting: while classes (and objects) are used to model data values to be symbolically manipulated, we add the concepts of agents and artifacts as coarse-grained abstractions to structure concurrent applications. Following an established tradition of papers in the object-orientation research contexts (see e.g. [7,8]), such an extension is developed in terms of a core calculus, namely, a very small language that focuses on the few aspects of interest while abstracting away from unnecessary language mechanisms. This should pave the way towards establishing technical analysis results over the bigger language, such as e.g. progress, soundness, and so on [22].

Differently from existing core calculi for object-oriented languages, however, our extension necessarily deals with concurrency aspects, calling for the joined exploitation of modelling techniques coming from the functional [22] and process-algebraic settings [3,12].

4.1 A subset of simpA

To develop our approach formally we first conceive a subset of the simpA framework neglecting some of its features. First of all, we do not consider activities with agenda, namely, activities have no guard condition and start executing as soon as they are scheduled. Then, there is no explicit notion of filtering linked to sensors; in our language the `use` action yields an action/sensor identifier, which can be later used to perceive events. Concerning artifacts, we only model fields and operations, the latter having a guard condition; so we do not consider observable properties and operation steps.

Based on this subset, we introduce an extension of the Java language with agent class definitions, providing fields and activities, and artifact class definitions, with fields and operations. Each simpA primitive is turned into a new instruction or operator: use actions over an artifact rely on syntax `art..operation(args)`, agents and artifacts access their fields by `this..field`, agents schedule activities by `this^activity(args)`, artifacts generate events by `event->`, and agents perceive them by expression `->sensor`. Though small, this programming framework is expressive enough to code interesting applications, like a typical dining philosophers example of coordination as shown in Figure 3—this example actually exploits Java arrays which are not modelled in this calculus, but this is a very orthogonal issue that do not harms the approach.

4.2 Syntax

This core calculus heavily relies on the settings of FJ calculus, the de-facto standard for modelling Java extensions [7].

We let metavariable `C` range over class names, `A` over artifact (class) names, `G` over agent (class) names, `f` over field names, `m` over method names, `a` over activity names, `o` over operation names, `x` over variables, `r` over internal references (to agents and artifacts) and `s` over action/sensor unique identifiers. As usual, we abbreviate "$\overline{\texttt{e}}$" for "$\texttt{e}_1, \ldots, \texttt{e}_n$", "$\overline{\texttt{T}}\ \overline{\texttt{x}}$" for "$\texttt{T}_1\,\texttt{x}_1, \ldots, \texttt{T}_n\,\texttt{x}_n$", "$\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}$" for "$\texttt{T}_1\,\texttt{f}_1\texttt{;} \ldots \texttt{;}\,\texttt{T}_n\,\texttt{f}_n$", and the like ($n \geq 0$). If $\overline{\texttt{e}}$ is a list of expressions, we denote by $\texttt{e}_i$ its i^{th} element. The abstract syntax of the calculus is shown in Figure 4.

A type `T` can be a class name `C`, or a reference type for an artifact `A` or an agent `G`. Note that classes are not considered as reference types for they are handled functionally as in FJ, that is, no side effect mechanism is implemented; therefore, their value is constant.

A program is formed by a list of definitions, of classes, agents and artifacts. A class definition includes its name, a list of fields $\overline{\texttt{f}}$ (each with its type $\texttt{T}_i$), and a list of methods (class inheritance is not modelled). Methods have a return type `T`, a name `m` and arguments $\overline{\texttt{T}}\ \overline{\texttt{x}}$, and their body is just a return statement. An agent (class) definition has a name `G`, a list of fields representing its state, and a list of activities $\overline{\texttt{Act}}$ representing its autonomous behaviour. An activity has a name `o`, arguments, and its body is an expression, treated as a statement. An artifact (class) definition has a name `A`, a list of fields representing its state, and a list of operations $\overline{\texttt{Op}}$ representing the service it provides. An operation has a name `o`, arguments, a guard expression, and its body is an expression, treated as a statement.

Expressions `e` 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 current method/operation/activity); literal `this` is handled as a special kind of variable. 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(`$\overline{\texttt{e}}$`)`). The latter expression is used for either objects, artifacts and agents.

```
class Chops{         // OO Model for a set of possibly-available chops
   boolean[] chops;
   boolean getChop(int pos){
      return chops[pos];
   }
   boolean setChop(int pos){
      this.chops[pos]=true;
   }
   boolean resetChop(int pos){
      this.chops[pos]=false;
   }
   boolean chopsAvailable(int left,int right){
      return chops[left] && chops[right];
   }
}

artifact Table {    // Artifact for synchronising access to chops
   Chops chops;

   operation getChops(int left, int right) guard chops.chopsAvailable(left,right){
      this.chops.resetChop(left);
      this.chops.resetChop(right);
      <-"chops_acquired";  // generating an event
   }
   operation releaseChops(int left, int right){
      this.chops.setChop(left);
      this.chops.setChop(right);
      <-"chops_released";  // generating an event
   }
}

agent Philosopher{   // An agent simply using the Table
   int leftChop;
   int rightChop;
   A table;
   S sensor;  // Local sensor

   activity main() {
      this..sensor=table..getChops(leftChop,rightChop);  // getting chops
      ->this..sensor;                                    // perceiving completion
      this^eating();                                     // new activity
   }
   activity eating(){
      ...
      this.sensor=table..releaseChops(leftChop,rightChop); // releasing chops
      ->this..sensor;                                      // perceive completion
      this^thinking();                                     // new activity
   }
   activity thinking() {
      ...
      this^main();
   }
}

agent Booter{       // An agent that boots the system
   Table artifact;
   activity main(){  // creating the table and agents
      this..artifact=new Table(new Chops(new boolean[]{true,true,true}));
      new Philosopher(this..artifact,0,1);
      new Philosopher(this..artifact,1,2);
      new Philosopher(this..artifact,2,0);
   }
}
```

Fig. 3. Code for the Dining Philosophers Application

Other expressions are included to model primitives of the simpA framework as new language mechanisms. Inside agents and artifacts, fields are accessed by syntax `e..f`, and are updated by syntax `e..f=e'` (`e'` evaluates to the new value). Similarly, an agent can use an artifact `e` (i.e., expression `e` evaluates to the artifact reference) by expression `e..o(`$\overline{\mathtt{e}}$`)`, specifying operation `o` and arguments $\overline{\mathtt{e}}$, and yielding an action unique reference. Expression `->e` is used to perceive an event corresponding to action reference `e`, `e^a(`$\overline{\mathtt{e}}$`)` to schedule activity `a` in the agent (obtained by evaluating) `e`, and finally `e->` is used to generate an event with content

R ::=	G | A	Reference Types
T ::=	R | C	Types
L ::=	class C{$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{Meth}}$}	Class definition
	| agent G{$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{Act}}$}	Agent definition
	| artifact A{$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{Op}}$}	Artifact definition
Act ::=	activity a($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){e;}	Activity definition
Op ::=	operation o($\overline{\texttt{T}}$ $\overline{\texttt{x}}$) guard e{e;}	Operation definition
Meth ::=	T m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){return e;}	Method definition
e ::=		Expressions/statements
	e;e	sequential composition
	| x	variable
	| e.f	class-field access
	| e.m($\overline{\texttt{e}}$)	method invocation
	| new T($\overline{\texttt{e}}$)	instance creation
	| e..f	agent/artifact-field access
	| e..f=e	agent/artifact-field update
	| e..o($\overline{\texttt{e}}$)	use
	| ->e	event sensing
	| e^a($\overline{\texttt{e}}$)	activity scheduling
	| e->	event signalling
	| r	reference
	| s	action identifier

Fig. 4. Syntax

e. Expressions include also references r and identifiers s.

In developing such a syntax, we made a number of assumptions to trade off simplicity of the calculus and completeness of features with respect to the simpA framework subset. First of all, the above syntax actually allows more programs than a parser for the language should accept—an actually typical scenario, as happen e.g. for the Java language. On the one hand, this allows for a compact and simpler representation of syntax, on the other hand, we do this to take into account also expressions temporarily produced during computation, which cannot be expressed in a program. Basically, further constraints to be applied to a valid program includes the fact that references r and s cannot appear in the surface language; that the receiver of a field access, field assignment, and activity schedule should be variable this; scheduling, use, and sensing can occur only in agents, whereas event generation can occur only in artifacts.

We abuse operator "$\in$" using it for syntactic structure inclusion, writing e.g. "Meth $\in$ C" to mean method Meth is included in the definition of class C. Syntax e[e'/x] denotes the expression obtained from e by substituting all occurrences of variable x with e', while $\overline{\texttt{e}}[i \mapsto \texttt{e}]$ represents list $\overline{\texttt{e}}$ after substituting its i^{th} element

$\mathtt{v},\mathtt{w}$	$::= \mathtt{r} \mid \mathtt{s} \mid \mathtt{new\ C}(\overline{\mathtt{v}})$	Values
$\mathtt{M}$	$::= 0$	empty configuration
	$\mid (\mathtt{M} \mid \mathtt{M})$	parallel composition
	$\mid \mathtt{r{=}R}(\overline{\mathtt{v}})$	agent or artifact
	$\mid \mathtt{r{:}a}(\overline{\mathtt{v}})\{\mathtt{e;}\}$	agent activity
	$\mid \mathtt{r(s){:}o}(\overline{\mathtt{v}})\langle\mathtt{e}\rangle\{\mathtt{e;}\}$	artifact operation
	$\mid \langle \mathtt{s,v} \rangle$	event

Fig. 5. Values and Configurations

with e.

4.3 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. This is expressed in our calculus in terms of a multiset of elements including the different stateful entities and processes living in the multiagent systems at a given time. We first introduce the concept of a value v, representing as usual the possible outcome of the evaluation of an expression; then introduce the concept of a configuration M: transitions over configurations are used to define operational semantics. The syntax of values and configurations is shown in Figure 5.

Values are either references (to artifacts, agents, or actions/sensors) or objects, namely, new C($\overline{\mathtt{v}}$) is an instance of class C with its fields filled with values $\overline{\mathtt{v}}$, orderly. We introduce syntactic sugar for two values **t** and **f**, which are shorthand for objects new True() and new False(), where True and False are library classes—the definition of such classes is avoided for simplicity. A configuration is a multiset of elements, where: r=R($\overline{\mathtt{v}}$) represents agent of artifact with reference r, class name R, and current value of fields $\overline{\mathtt{v}}$; term r:a($\overline{\mathtt{v}}$){e;} represents the execution of activity a of agent with reference r, with arguments $\overline{\mathtt{v}}$, and body execution state e; similarly r(s):o($\overline{\mathtt{v}}$)⟨e⟩{e_b;} represents operation o of artifact r executed with arguments $\overline{\mathtt{v}}$, guard value e and body e_b; and finally ⟨s,v⟩ represents action with reference s and content v.

4.4 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 [22,20]. Their syntax is expressed in Figure 6.

Since a context has one hole $[\![]\!]$ in it, syntax $\mathbb{E}[\![\mathtt{e}]\!]$ denotes context $\mathbb{E}[\![]\!]$ after substituting its hole with expression e, and similarly for the other kinds of context. Contexts are useful to identify what is the next sub-expression that needs to be evaluated, imposing a proper evaluation order, abstracting away from the other

$$
\begin{array}{lll}
\mathbb{E} ::= [\![]\!]\texttt{;e} \mid [\![]\!]\texttt{.f} \mid [\![]\!]\texttt{.m}(\overline{\texttt{e}}) \mid \texttt{v.m}(\overline{\texttt{v}}\ [\![]\!]\ \overline{\texttt{e}}) & & \text{Expression context} \\
\quad \mid \texttt{new T}(\overline{\texttt{v}}\ [\![]\!]\ \overline{\texttt{e}}) \mid [\![]\!]\texttt{..f} \mid [\![]\!]\texttt{..f=e} \mid \texttt{v..f=}[\![]\!] & & \\
\quad \mid [\![]\!]\texttt{..o}(\overline{\texttt{e}}) \mid \texttt{v..o}(\overline{\texttt{v}}\ [\![]\!]\ \overline{\texttt{e}}) \mid \texttt{->}[\![]\!] \mid [\![]\!]\texttt{->} & & \\
\mathbb{B} ::= \texttt{r:a}(\overline{\texttt{v}})\{[\![]\!]\texttt{;}\} \mid \texttt{r(s):o}(\overline{\texttt{v}})\langle\texttt{t}\rangle\{[\![]\!]\texttt{;}\} & & \text{Body context} \\
\mathbb{G} ::= \texttt{r(s):o}(\overline{\texttt{v}})\langle[\![]\!]\rangle\{\texttt{e;}\} & & \text{Guard context} \\
\mathbb{K} ::= \mathbb{B} \mid \mathbb{G} & & \text{Configuration context}
\end{array}
$$

Fig. 6. Evaluation contexts

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

$$
\frac{\mathbb{K}[\![\texttt{e}]\!] \longrightarrow \mathbb{K}[\![\texttt{e'}]\!]}{\mathbb{K}[\![\mathbb{E}[\![\texttt{e}]\!]]\!] \longrightarrow \mathbb{K}[\![\mathbb{E}[\![\texttt{e'}]\!]]\!]} \quad [\text{EVAL}]
$$

$$
\mathbb{K}[\![\texttt{v;e}]\!] \longrightarrow \mathbb{K}[\![\texttt{e}]\!] \quad [\text{SEQ}]
$$

$$
\mathbb{B}[\![\texttt{v}]\!] \longrightarrow \texttt{0} \quad [\text{LAST}]
$$

$$
\frac{\overline{\texttt{T}}\ \overline{\texttt{f}} \in \texttt{C}}{\mathbb{K}[\![\texttt{new C}(\overline{\texttt{v}})\texttt{.f}_\texttt{i}]\!] \longrightarrow \mathbb{K}[\![\texttt{v}_\texttt{i}]\!]} \quad [\text{FIE}]
$$

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

Fig. 7. Operational semantics: evaluation and oo layer

parts of the program where such a sub-expression is inserted in. When an expression matches $\mathbb{E}[\![\texttt{e}]\!]$, it means that e is the next sub-expression that needs to be evaluated: the idea is that in case e evaluates to v, then the whole expression $\mathbb{E}[\![\texttt{e}]\!]$ evaluates to $\mathbb{E}[\![\texttt{v}]\!]$. The way the syntax of $\mathbb{E}$ is structured guarantees e.g. that a receiver is evaluated before the method arguments are evaluated, and that such arguments are evaluated from left to right, and so on. For operations, $\mathbb{K}$ is structured so that guard evaluation proceeds until reaching value **t**, after that the body of the operation can be executed.

4.5 Operational Semantics

Operational semantics is defined as a non-deterministic transition system over configurations, whose rules are reported in Figure 7 and 8.

Figure 7 deals with evaluation of subexpressions, sequential composition, and typical object-oriented operations: it hence models aspects which are mostly orthogonal to agents and artifacts. Rule [PAR] sets an interleaved parallelism model for this calculus, saying that any system subpart `M'` can spontaneously evolve to `M''`. The only form of global awareness that we really require is due to predicate "`fresh r`" (or `s`) which is used to generate a globally-new reference identifier `r`—as used in other rules. Rule [EVAL] is used to propagate evaluations inside contexts, namely, inside a context $\mathbb{K}[\![]\!]$ representing some code execution, an expression/statement of the kind $\mathbb{E}[\![\texttt{e}]\!]$ should allow subsexpression `e` to be evaluated. Note that this rule allows us to forget about contexts of the kind $\mathbb{E}[\![]\!]$ in all other rules, that is, we can always suppose that expressions are in simple forms ready to be evaluated. Rule [SEQ] is a first remarkable example: if we have a context $\mathbb{K}[\![\texttt{v;e}]\!]$ it means the first statement of the block have already been executed, hence we can move to $\mathbb{K}[\![\texttt{e}]\!]$. Similarly, by rule [LAST] we deal with the last instruction in a body: if we have a context $\texttt{B}[\![\texttt{v}]\!]$, it can be moved to the empty configuration 0.

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 same rules in FJ). In both cases the receiver is a fully-evaluated expression of the kind "`new C(`$\overline{\texttt{v}}$`)`", representing an object of class `C` with values $\overline{\texttt{v}}$ in their fields. Accordingly, [FIE] retrieves the i^{th} field, while [INVK] retrieves the expression body `e` after substituing `this` with "`new C(`$\overline{\texttt{v}}$`)`", and formal parameters $\overline{\texttt{x}}$ with actual arguments $\overline{\texttt{e}}$.

Figure 8 deals with expressions and instructions, which are key aspects of simpA. Rule [ART] handles creation of an artifact by expression "`new A(`$\overline{\texttt{v}}$`)`": this causes the creation of item `r=A(`$\overline{\texttt{v}}$`)` in the configuration, exploiting a new reference `r` which is returned as evaluation result. Thanks to rules [GET] and [SET] we can access and modify fields over an artifact—or an agent. The main difference with respect to rule [FIE] is that in [GET] the receiver is a global reference `r`, which is used to lookup for the actual fields content $\overline{\texttt{v}}$; similarly, rule [SET] updates the state associated to `r`.

The instantiation of an agent by rule [AGN] creates a new reference as in rule [ART], but moreover it schedules the default activity called `main` by inserting an item `r:main(){e[r/this]}`. In the general case, scheduling handles an expression `r^a(`$\overline{\texttt{w}}$`)` by rule [SCHED], which looks for the agent class `G` associated to reference `r`, retrieves the definition of `a`, and then creates a proper item in the configuration. Rule [USE] tights together agents and artifacts, handling expression `r..o(`$\overline{\texttt{v}}$`)`; after retrieving the artifact class `A` and a definition for operation `o`, an action identifier `s` is also generated and returned, and an item for the operation is created in the configuration. Note that initially that operation is guarded by expression `e`, which by rule [EVAL] will be eventually evaluated: when such an evaluation returns **t**, context $\mathbb{K}[\![]\!]$ automatically allows the operation body to be executed. If it evaluates to **f** instead, it means the operation cannot be executed yet, hence rule [GUARD] substitutes **f** with the original guard expression `e` which gets evaluated again. The idea here is that in this model, guards keep evaluating until the overall system configuration is such that they reach value **t**.

$$\frac{\texttt{r fresh}}{\mathbb{K}[\![\texttt{new A}(\overline{\texttt{v}})]\!] \longrightarrow \texttt{r=A}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r}]\!]} \quad [\text{ART}]$$

$$\frac{\overline{\texttt{T}}\ \overline{\texttt{f}} \in \texttt{R}}{\texttt{r=R}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r..f}_i]\!] \longrightarrow \texttt{r=R}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{v}_i]\!]} \quad [\text{GET}]$$

$$\frac{\overline{\texttt{T}}\ \overline{\texttt{f}} \in \texttt{R} \qquad \overline{\texttt{w}} = \overline{\texttt{v}}[i \mapsto \texttt{w}]}{\texttt{r=R}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r..f}_i\texttt{=w}]\!] \longrightarrow \texttt{r=R}(\overline{\texttt{w}}) \mid \mathbb{K}[\![\texttt{w}]\!]} \quad [\text{SET}]$$

$$\frac{\texttt{r fresh} \qquad \texttt{main()\{e;\}} \in \texttt{G}}{\mathbb{K}[\![\texttt{new G}(\overline{\texttt{v}})]\!] \longrightarrow \texttt{r=G}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r}]\!] \mid \texttt{r:main()\{e[r/this]\}}} \quad [\text{AGN}]$$

$$\frac{\texttt{a}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{e';\}} \in G}{\texttt{r=G}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r\^{}a}(\overline{\texttt{w}})]\!] \longrightarrow \texttt{r=G}(\overline{\texttt{v}}) \mid \mathbb{K}[\![\texttt{r}]\!] \mid \texttt{r:a}(\overline{\texttt{w}})\{\texttt{e[r/this]}[\overline{\texttt{w}}/\overline{\texttt{x}}]\}} \quad [\text{SCHED}]$$

$$\frac{\texttt{s fresh} \qquad \texttt{o}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{guard e \{e';\}} \in A}{\mathbb{K}[\![\texttt{r..o}(\overline{\texttt{v}})]\!] \mid \texttt{r=A}(\overline{\texttt{w}}) \longrightarrow \mathbb{K}[\![\texttt{s}]\!] \mid \texttt{r=A}(\overline{\texttt{w}}) \mid \texttt{r(s):o}(\overline{\texttt{v}})\langle e\rangle\texttt{\{e'[r/this]}[\overline{\texttt{w}}/\overline{\texttt{x}}]\texttt{\}}} \quad [\text{USE}]$$

$$\frac{\texttt{o}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{guard e\{e';\}} \in A}{\texttt{r=A}(\overline{\texttt{w}}) \mid \texttt{r(s):o}(\overline{\texttt{v}})\langle\texttt{f}\rangle\texttt{\{e'';\}} \longrightarrow \texttt{r=A}(\overline{\texttt{w}}) \mid \texttt{r(s):o}(\overline{\texttt{v}})\langle\texttt{e[r/this]}[\overline{\texttt{w}}/\overline{\texttt{x}}]\rangle\texttt{\{e'';\}}} \quad [\text{GUARD}]$$

$$\texttt{r(s):o}(\overline{\texttt{v}})\langle\texttt{t}\rangle\{\mathbb{E}[\![\texttt{v->}]\!]\} \longrightarrow \texttt{r(s):o}(\overline{\texttt{v}})\langle\texttt{t}\rangle\{\mathbb{E}[\![\texttt{v}]\!]\} \mid \langle\texttt{s,v}\rangle \quad [\text{GEN}]$$

$$\mathbb{K}[\![\texttt{->s}]\!] \mid \langle\texttt{s,v}\rangle \longrightarrow \mathbb{K}[\![\texttt{v}]\!] \quad [\text{PER}]$$

Fig. 8. Operational semantics: agents and artifacts layer

5 Related Work

The artifact abstraction is a generalisation of *coordination artifacts*—i.e. artifacts encapsulating coordination functionalities—introduced in [16]. Actually, from the perspective of coordination models [17], coordination artifacts can be framed as *coordination media*, with some further or specific foundational properties that make their adoption akin with the notion of agent: the main one concerns the fact that agents encapsulates their control flow(s), which cannot be blocked then by the coordination media when primitives are executed (such as the *in* primitive in Linda tuple spaces). Such an issue is not just a conceptual or philosophical aspect, it is a fundamental aspect when the engineering of complex—concurrent, distributed—systems is considered. The interested reader can be found more on these issues in [15].

In A&A, artifacts are the basic building blocks that can be used to engineer the

working environments where agents are situated: agent environment then play a fundamental role here in engineering the overall MAS as first-order entity that can be designed so as to encapsulate some kind of responsibility (functionality, service). This perspective is explored in several research works appeared recently in MAS literature: a survey can be found here [21].

The extension of the OO paradigm toward concurrency—i.e., object-oriented concurrent programming (OOCP)—has been (and indeed is still) one of the most important and challenging themes in the OO research. Accordingly, a quite large amount of theoretical results and approaches have been proposed since the beginning of the 80's; it is not possible to report here a full list of all the approaches: the interest reader is forwarded to surveys such as [5,23,2]. Among the main examples, *active objects* [9] and *actors* [1] have been the root of entire families of approaches.

The approach proposed in this paper shares the aim of actors and active-objects, introducing a general-purpose abstraction layer to simplify the development of concurrent applications. Actually, this perspective is common in some recent approaches extending OO with modern concurrency abstractions, like Polyphonic C# [4]. Differently from actor-based approaches, in A&A and simpA also the passive components of the systems are modelled as first-class entities (the artifacts), besides the active parts (actors in actor-based systems). Differently from active-object-based approaches—where typically active-objects are objects with further capabilities—, in simpA a strong distinction between active and passive entities is promoted: agents and artifacts have completely different properties, with a clear distinction at the design level of their role, i.e. encapsulating pro-active / task-oriented behaviour (agents) and passive / function-oriented behaviour (artifacts). As Polyphonic C# introduces a very elegant support—grounded on Join calculus theory—for the synchronisation and coordination of multi-threaded applications, the objective of our approach is more extensive: we introduce an abstraction layer which aims at providing an effective support for tackling not only synchronisation and coordination issue, but also the engineering of passive and active parts of the application, avoiding the use of low level mechanisms such as threads.

Summing up, in our perspective the novelty of A&A programming model—and of simpA as a first concrete implementation over Java—is the identification of a high-level, human-inspired level of abstraction for multiprogramming and engineering complex software systems, beyond the one typically adopted by existing approaches on extensions of objects, processes, threads, and a-like.

6 Conclusion

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—quoting Sutter and Larus [19]—"[...]higher-level abstractions that help build concurrent programs, just as object-

oriented abstractions help build large componentised programs.".

Along this line, in this paper we presented simpA, a library extension over the basic Java platform that aims at simplifying the development of complex (concurrent) applications by introducing a high-level agent-oriented abstraction layer over the OO layer, and a first core calculus formalising a subset of simpA. This calculus is currently in the form of a formalisation attempt, based on previous works on core calculi for object oriented languages such as FJ, and process algebras. Based on it, several research directions have to be explored to make it a suitable basis for analysing properties of programs exploiting agents and artifacts. First of all, it would be interesting to introduce a type system that could be used to guarantee well-formedness properties of simpA programs. Then, we can state progress results based on deadlock avoidance. Moreover, the calculus itself should be extended to deal with a significantly larger language, eventually covering all features related to agents and artifacts.

References

[1] Agha, G., "Actors: a model of concurrent computation in distributed systems," MIT Press, Cambridge, MA, USA, 1986.

[2] Agha, G., P. Wegner and A. Yonezawa, editors, "Research directions in concurrent object-oriented programming," MIT Press, Cambridge, MA, USA, 1993.

[3] Baeten, J. C. M., J. A. Bergstra and J. W. Klop, *Decidability of bisimulation equivalence for processes generating context-free languages*, in: *PARLE, Parallel Architectures and Languages Europe, Volume I: Parallel Languages*, Lecture Notes in Computer Science **259** (1987), pp. 94–111.

[4] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C#*, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 769–804.

[5] Briot, J.-P., R. Guerraoui and K.-P. Lohr, *Concurrency and distribution in object-oriented programming*, ACM Comput. Surv. **30** (1998), pp. 291–329.

[6] Ferber, J., "Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence," Addison Wesley, 1999.

[7] Igarashi, A., B. C. Pierce and P. Wadler, *Featherweight Java: A minimal core calculus for Java and GJ*, ACM Transactions on Programming Languages and Systems **23** (2001), pp. 396–450.

[8] Igarashi, A. and M. Viroli, *Variant parametric types: A flexible subtyping scheme for generics*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 795–847.

[9] Lavender, R. G. and 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.

[10] Lea, D., *The java.util.concurrent synchronizer framework*, Sci. Comput. Program. **58** (2005), pp. 293–309.

[11] Lieberman, H., *The continuing quest for abstraction.*, in: D. Thomas, editor, *ECOOP*, Lecture Notes in Computer Science **4067** (2006), pp. 192–197.

[12] Milner, R., "Communicating and Mobile Systems: The π-calculus," Cambridge University Press, 1999.

[13] Nardi, B. A., "Context and Consciousness: Activity Theory and Human-Computer Interaction," MIT Press, 1996.

[14] Omicini, A., A. Ricci and M. Viroli, *Agens Faber: Toward a theory of artefacts for MAS*, Electronic Notes in Theoretical Computer Sciences **150** (2006), pp. 21–36.

[15] Omicini, A., A. Ricci and M. Viroli, *Coordination artifacts as first-class abstractions for MAS engineering: State of the research*, in: A. F. Garcia, R. Choren, C. Lucena, P. Giorgini, T. Holvoet and A. Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, LNAI **3914**, Springer, 2006 pp. 71–90, invited Paper.

[16] Omicini, A., A. Ricci, M. Viroli, C. Castelfranchi and L. Tummolini, *Coordination artifacts: Environment-based coordination for intelligent agents*, AAMAS'04 **1** (2004), pp. 286–293.

[17] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, Advances in Computers **46** (1998), pp. 329–400.

[18] Ricci, A., M. Viroli and A. Omicini, *Programming MAS with artifacts*, in: R. P. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, LNAI **3862**, Springer, 2006 pp. 206–221.

[19] Sutter, H. and J. Larus, *Software and the concurrency revolution*, ACM Queue: Tomorrow's Computing Today **3** (2005), pp. 54–62.

[20] Viroli, M., *A core calculus for correlation in orchestration languages*, Journal of Logic and Algebraic Programming **70** (2007), pp. 74–95, Special Issue on Web Services and Formal Methods.

[21] Weyns, D. and H. V. D. Parunak, editors, Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Environment for Multi-Agent Systems **14(1)**, Springer Netherlands, 2007.

[22] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.

[23] Yonezawa, A. and M. Tokoro, editors, "Object-oriented concurrent programming," MIT Press, Cambridge, MA, USA, 1986.