simpA: A Simple Agent-Oriented Java Extension for Developing Concurrent Applications

Alessandro Ricci, Mirko Viroli, and Giulio Piancastelli

DEIS, Alma Mater Studiorum – Università di Bologna, Italy
{a.ricci,mirko.viroli,giulio.piancastelli}@unibo.it

Abstract. More and more aspects of concurrency and concurrent programming are becoming part of mainstream programming and software engineering, as a result of several factors, such as the widespread availability of multi-core / parallel architectures and Internet-based systems. Java has been one of the first mainstream languages providing a first-class native support for multi-threading, with basic low level *fine-grained* concurrency mechanisms. Besides this fine-grained support to concurrency, the identification of higher-level—more *coarse-grained*—support is important as soon as programming and engineering complex concurrent applications is considered, helping to bridge the gap between system design, implementation and testing.
Accordingly, in this paper we present simpA, a library-based extension of Java which introduces a high-level coarse-grained support to prototyping complex, multi-threaded / concurrent applications: Java programmers are provided with an *agent-oriented* abstraction layer on top of the basic OO layer to organize and structure applications.

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, Internet-based computing and Service-Oriented Architectures / Web Services are further main driving factors introducing concurrency issues in the context of a large class of applications and systems, no more related only to specific and narrow domains, such as high-performance scientific computing.

As noted in [18], if on the one hand concurrency has been studied for about 30 years in the context of computer science fields such as programming languages and software engineering, on the other hand this research has not had a strong impact on mainstream software development. As a main example, Java has been one of the first mainstream languages providing a first-class native support for multi-threading, with basic low level concurrency mechanisms. Such a support has been recently extended by means of a new library added to the JDK with classes that implement well-known and useful higher-level synchronisation mechanisms such as barriers, latches, semaphores, providing a *fine-grained* and efficient control on concurrent computations [9]. Besides this fine-grained support to concurrency, it appears more and more important to introduce higher-level abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [18]. Agents and multi-agent systems (MASs)—in their most general characterisation—are very promising abstractions for this purpose, natively capturing and modelling decentralisation of control, concurrency of activities, and interaction / coordination of activites: therefore, they can be considered a good candidate for defining a paradigm for mainstream concurrent programming, beyond OO.

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 programming and software engineering as a novel basic approach for modelling and engineering multi-agent systems [15, 12]. *Agents* and *artifacts* are the basic high-level coarse-grained abstractions available in A&A (and simpA): the former is used in A&A to model (pro)-active and activity-oriented components of a system, encapsulating the logic and control of such activities, while the latter is used to model function-oriented components of the system, used by agents to support their activities.

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 provides some discussion about the overall 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 [10]:

> "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 in general, in providing 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 theories such as Activity Theory (AT) [11]. This context is taken here as a reference example of a complex system, where multiple concurrent activities are carried on in a coordinated manner, interacting within some kind of working environment: humans do work concurrently and cooperatively in the context of the same activities, interacting both directly, 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

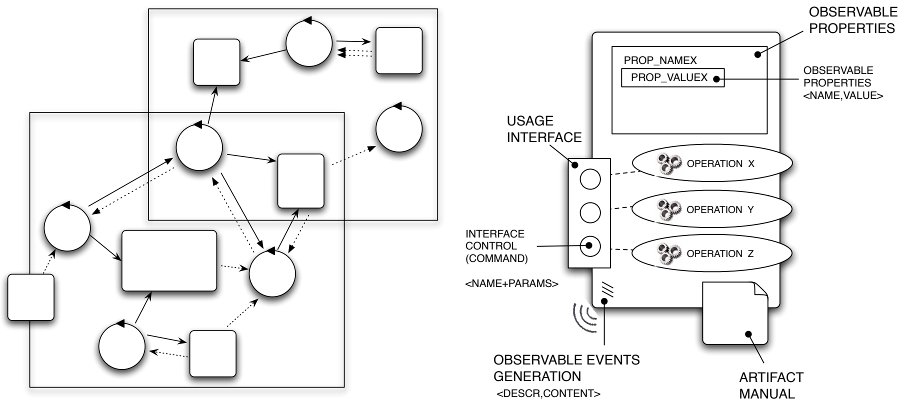


Fig. 1. *(Left)* An abstract representation of an application according to the A&A programming model, as a collection of agents (circles) sharing and using artifacts (squares), grouped in workspaces. *(Right)* An abstract representation of an artifact, with in evidence the usage interface, with commands (control) to trigger the execution of operations, the observable properties and the manual.

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, a coffee machine) or instruments mediating and coordinating collective activities (e.g. a blackboard, a calendar, a task scheduler, etc).

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 (pro-)active 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, either individually or collectively. 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 agents and artifacts, and can be used to structure the overall sets of entities, defining a topology of the working environment and providing a way to frame the interaction inside it (see Fig. 1, left).

2.1 Agent and Artifact Abstractions: Core Properties

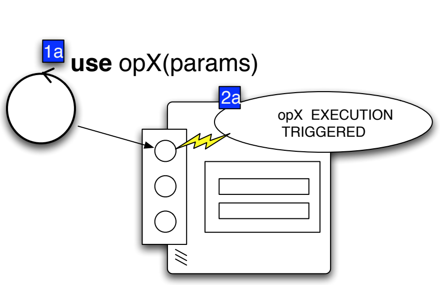
In A&A the term "agent" is used in its etymological meaning of an entity "who acts", i.e. whose compuational behaviour accounts for performing *actions* in some kinds of environment and getting information back in terms of *perceptions*. In A&A agents' actions and perceptions concern in particular the *use* of artifacts and direct communication with other agents. 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 state-ful entity, each agent has a *long-term memory*, used to store data and information needed for its overall work, and a *short-term memory*, as a working memory used to store temporary information useful when executing single activities. An agent can carry on multiple activities concurrently, and each activity in execution defines a context, as a local scope for storing information related to the specific activity, contextualising the execution of actions and perceiving events from the agent environment. Here it's worth remarking that the notion of agent in simpA is not meant to be comparable with models and architectures as typically found in the context of intelligent agents or cognitive agent platforms, such as Jason, 2APL, 3APL, JACK, JADEX or alike: here the objective is not maximising flexibility and autonomy so as to play in unpredictable and complex environments, but having a basic simple abstraction which would make it natural and straightforward to design and program complex active behaviours, providing some strong encapsulation properties for state and control of activities.

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 artifact *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 agents can use to trigger and control operation execution (like the control panel of a coffee machine), each one identified by a label (typically equals to the operation name to be triggered) and a list of input parameters. In this *use* interaction 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. Besides the controls for triggering the execution of operation, an artifact can define some *observable properties*, as variables whose value can be inspected by agents dynamically, without necessarily executing operations on it (like the display of a coffee machine).

2.2 Agent-Artifact Interaction: Use and Observation

The interaction between agents and artifacts 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 machine represents the usage interface, while the displays used to show the state of the machine represent artifact ob-



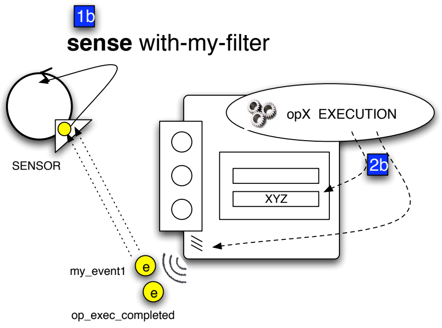


Fig. 2. Abstract representation of an artifact (on the left), and of an agent using an artifact, by triggering the execution of on operation (center, step 1a) and observing the related events generated (right, step 1b)

servable 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 and control the execution of an operation over an artifact. The observable events possibly generated by the artifact executing an operation are collected by agent *sensors*, that are those parts of the agent (body) 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 to explicitly retrieve / 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—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 with 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, namely, as basic high-level building blocks to program concurrent applications. This approach contrasts most existing ones modifying object-oriented abstractions (classes, objects, methods) to model concurrency aspects—such as e.g. [2]. 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 that 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 compiled 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. simpA technology is open-source and is available for download at simpA web site[1].

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 here as a reference to exemplify the programming of agents and artifacts. The application creates a simple `Cafeteria` workspace, composed by 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 selecting the type of drink (coffee or tea) first, then for making the drink. Then, while making the drink, it provides a usage interface to adjust the sugar level and possibly to stop the operation (for 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 production lasts too long.

A simpA application is typically booted by setting up the workspace(s), creating an initial set 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. This example is part the basic examples provided in simpA distribution, available on simpA web site.

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

[1] `http://www.alice.unibo.it/simpa`

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, activities in particular, are mapped into class elements, suitably annotated. 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 is defined with a long-term memory composed by `drink1` and `drink2` variables, and by four atomic activities: `makeOneCoffee`, `makeOneTea`, `deliverBoth`, `deliverJustCoffee`.

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, the todo in the agenda are executed as soon as their pre-conditions hold. 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 behaviour of agents when executing some activity. Todo preconditions are expressed as a boolean expression, with and / or connectors (represented by `,` and `;` symbols, respectively) over a basic set of predefined predicates. Essentially, the 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 then used also to help the coordination of the various sub-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 todo: 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 coffe (`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. Actually, 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 artifacts are 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 class elements, suitably annotated. 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*, 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

```
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() throws Exception {
    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() throws Exception {
    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");
      throw new ActivityFailed();
    }
  }

  @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. A exemplifying case of simpA application, composed at runtime by a single `Waiter` agent using two instances—`cmOne` and `cmTwo`—of the `CoffeeMachine` artifact

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 operation (step) guarded. 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. 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 by 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 typically should 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 events is observed by all the agents that are *focussing* (observing) the artifact. 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, specifying 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 possibly to stop the process for having shorter drinks. Also, it has two observable properties, selection which reports the type of the drink currently selected, and sugarLevel which reports current level of the 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 it is executed 3 seconds after triggering. The second one is executed as soon as makingStopped guard is evaluated to true. This can happens if the agent user executed the stop operation while the coffee machine is making the coffee. In both cases, step execution accounts for releasing the drink, by signaling a proper event of the type coffee_ready or tea_ready, updating the observable properties value and switching to the idle state.

Some other artifact features are not described in detail here for lack of space. Among the other we mention: *linkability*—which accounts for dynamically composing artifacts together through *link interfaces*, which are interfaces with operations that are meant to be invoked (*linked*) by other artifacts—and *artifact manual*—which concerns the possibility to equip each artifact with a document, written by the artifact programmer, containing a formal machine-readable semantic-based description of artifact functionality and usage instructions (operating instructions). The interested reader is forwarded to the documentation available at simpA web site.

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—through agent sensors—the observable events generated by the artifact.

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 succeeds, the return parameter returned by `use` is the operation unique identifier. If the action execution fails—because, for instance, the interface control specified is not part of 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 a timeout is reached, optionally specified as a further parameter. As result of a successful execution of a `sense`, the event is removed from the sensor and a perception related to that event—represented by an object instance of the class `Perception`—is returned. If no perception are sensed for the duration of time specified, the action generates an exception of the kind `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 in observing 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 having triggered the `make` operation, then a memo `tea_not_ready` is taken and the activity fails (by means of the generation of an exception).

Actually, simpA provides a basic support also for agent direct communication, with a `tell(ReceiverId,Msg)` primitive to send a message to another agent, and a `listen(SensorId,Filter)`—analogous to focus primitive—to specify sensors to be used to collect the messages). So also for direct communication, sensors and sensing primitives are exploited to collect and be aware of perceptions, in this case related to the arrival of a message.

4 Discussion

The main objective of simpA is to simplify the prototyping of complex applications involving elements of concurrency, by introducing high level abstractions on top of the basic Object-Oriented layer.

As a first benefit, the level of abstraction underlying the approach is meant to promote an agile design of the application and then to reduce the gap between such design and the implementation level. At the design level, by adopting a task oriented approach as typically promoted by agent-oriented methodologies [5], the task-oriented and function-oriented parts of the system are identified, driving to the definition of the agents and artifacts as depicted by the A&A model, and then to the implementation in simpA.

Then, the approach aims at providing agile but quite general means to organise and manage complex active and passive behaviours. For the former, the notion of activity and the hierarchical activity model adopted in the agent abstraction make it possible to describe in a quite synthetic and readable way articulated active behaviours, abstracting away from the complexity related to threads creation, management and coordination. Besides the notion of activity, the very notion of agent as the state-full entity responsible for activity execution strengthen the level of encapsulation adopted to structure active parts. For the latter, the model of artifact adopted allows the programmer to specify complex functionalities (operations) possibly shared and exploited by multiple agents concurrently, without the need to explicitly use lower-level Java mechanisms such as synchronised blocks or wait / notify synchronisation primitives. On the one side, mutual exclusion in accessing and modifying artifact inner state is guaranteed by having only one operation step in execution at a time. On the other side, possible dependencies between operations can be explicitly taked into the account by defining the operation (step) guards.

Finally, besides the individual component level, the approach has been conceived to simplify the development of systems composed by multiple agents that work together, coordinating their activities by exploiting suitable *coordination artifacts* [13]. More generally, the problems that are typically considered in the context of concurrent programming involving the interaction and coordination of multiple processes—examples are *producer-consumer*, *readers-and-writers*, *dining-philosophers*—can be naturally modelled in terms of agents and artifacts, providing solutions that in our opinion are more clear and "high-level" with respect to those mixing object-oriented abstractions—threads and low-level synchronisation mechanism—as in the case of Java. For instance, producers-consumers problems are naturally modelled in terms of producer and consumer agents sharing and exploiting a bounded buffer artifact; readers-and-writers problems in terms of reader and writer agents that use a suitably designed rw-lock coordination artifact to coordinate their access to a shared resource; dining-philosophers, in terms of a set of philosopher agents sharing and using a table, which encapsulates and enforces those coordination rules that make it possible to handle mutual exclusion in using chopsticks and to avoid deadlock situations. For the interested readers, these and other problems are included among the examples provided in simpA distribution, not reported here for lack of space.

5 Related Works

simpA is implemented on top of CARTAGO, a Java-based platform for programming artifact-based working environments for MAS [17]. While simpA introduces a specific (simple) programming model for programming agents, CARTAGO is focussed solely on artifacts—programming and API for agents to use them—, and conceptually it can be integrated with heterogeneous agent platforms—including cognitive agent platforms, extending them to support artifact-based environments. A first concrete example of such a possibility is described in [16], where CARTAGO is integrated with the *Jason* agent platform [3], enabling *Jason* agents to create, share and use artifacts.

The artifact abstraction is a generalisation of *coordination artifacts*—i.e. artifacts encapsulating coordination functionalities, introduced in [13]. 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 in [19]. simpA is implemented on top of CARTAGO, a Java-based platform for programming artifact-based working environments for MAS [17].

Among the agent-oriented extensions of Java we cite here JACK [6], and JADEX [14], which extend the basic Java platform in order to support the programming of intelligent agents, based on the BDI architecture and the FIPA standards. These approaches—as most of the other cognitive agent programming platforms—are typically targeted to the engineering of distributed intelligent systems for complex application domains.

Among the agent-oriented Java-based platforms based on flat Java, with no extension of the basic language / platform, we mention here JADE [7]. JADE provides a general-purpose middleware that complies with the FIPA specifications for developing peer-to-peer distributed agent based applications. A main conceptual and practical difference between simpA and JADE concerns the high-level first-class abstractions adopted to organise a software system: in JADE there are agents interacting by means of FIPA ACL, in simpA there are agents and artifacts. Then, besides the support for FIPA ACL, JADE adopts a behaviour-based programming model for programming agents. From this point of view, activities in simpA are similar to behaviours in JADE, with the main difference that in simpA the definition of structured activities composed by sub-activities is done declaratively by defining the activity agenda, while in JADE is done operationally, by creating and composing objects of specific classes.

Finally, 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 [4, 20]. Among the main examples, *active objects* [8] and *actors* [1] have been the root of entire families of approaches. The approach proposed in this paper shares the aim of actor and active-objects approaches, i.e introducing a general-purpose abstraction layer to simplify the development of concurrent applications. 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).

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 architecture—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 [18]—"[...]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 simple high-level agent-oriented abstraction layer over the OO layer. Future work will be devoted on finalising a formal model for simpA basic programmming model on the one side, and defining a full fledged simpA language and virtual machine on top of the Java platform.

References

1. Agha, G., "Actors: a model of concurrent computation in distributed systems," MIT Press, Cambridge, MA, USA, 1986.
2. Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C#*, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 769–804.

3. Bordini, R. and J. H ubner, *BDI agent programming in AgentSpeak using Jason*, in: F. Toni and P. Torroni, editors, *CLIMA VI*, LNAI **3900**, Springer, 2006 pp. 143–164.
4. Briot, J.-P., R. Guerraoui and K.-P. Lohr, *Concurrency and distribution in object-oriented programming*, ACM Comput. Surv. **30** (1998), pp. 291–329.
5. Iglesias, C., M. Garrijo and J. Gonzalez, *A survey of agent-oriented methodologies*, in: J. Müller, M. P. Singh and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence **1555** (1999), pp. 317–330.
6. *JACK intelligent agents.*, `http://www.agent-software.com/`.
7. *JADE platform*, `http://jade.tilab.com/`.
8. 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.
9. Lea, D., *The java.util.concurrent synchronizer framework*, Sci. Comput. Program. **58** (2005), pp. 293–309.
10. Lieberman, H., *The continuing quest for abstraction.*, in: D. Thomas, editor, *ECOOP*, Lecture Notes in Computer Science **4067** (2006), pp. 192–197.
11. Nardi, B. A., "Context and Consciousness: Activity Theory and Human-Computer Interaction," MIT Press, 1996.
12. 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.
13. 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.
14. Pokahr, A., L. Braubach and W. Lamersdorf, *Jadex: A BDI reasoning engine*, in: R. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, Kluwer, 2003 .
15. 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.
16. Ricci, A., M. Viroli and A. Omicini, *A general purpose programming model & technology for developing working environments in MAS*, in: M. Dastani, A. El Fallah Seghrouchni, A. Ricci and M. Winikoff, editors, *5th International Workshop "Programming Multi-Agent Systems" (PROMAS 2007)*, AAMAS 2007, Honolulu, Hawaii, USA, 2007, pp. 54–69.
17. Ricci, A., M. Viroli and A. Omicini, CArtAgO*: A framework for prototyping artifact-based environments in MAS*, in: D. Weyns, H. V. D. Parunak and F. Michel, editors, *Environments for MultiAgent Systems III*, LNAI **4389**, Springer, 2007 pp. 67–86.
18. Sutter, H. and J. Larus, *Software and the concurrency revolution*, ACM Queue: Tomorrow's Computing Today **3** (2005), pp. 54–62.
19. 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.
20. Yonezawa, A. and M. Tokoro, editors, "Object-oriented concurrent programming," MIT Press, Cambridge, MA, USA, 1986.