simpA-WS: A Simple Agent-Oriented Programming Model & Technology for Developing SOA & Web Services

Alessandro Ricci, Enrico Denti

***Abstract*— Service-Oriented Architecture (SOA) is more and more recognised by the industry as the reference blueprint for building inter-operable, distributed enterprise applications based on open standards such as Web Services (WS). In the current state-of-the-art, the programming models for engineering SOA systems proposed by the leading industries are essentially *component-based* – typically, rooted in object-oriented abstractions and technologies. On the side, such a choice benefits from the well-know advantages of component-based software engineering and from the maturity of the available technologies; on the other, however, the abstraction level provided is inadequate to model some fundamental SOA aspects – such as autonomy, control-uncoupling, data-driven interaction, activities – as first-class concepts. Such features can be modelled quite naturally by adopting an *agent-oriented* perspective.**

In this paper we describe simpA-WS, a Java-based framework for developing SOA/WS applications which adopts an agent-oriented programming model based on the general-purpose *Agents and Artifacts* meta-model (A&A). simpA-WS makes it possible to conceive, design and program services (and applications using services) as workspaces where ensemble of pro-active, activity-oriented entities (agents) work together by exploiting different kinds of passive function-oriented entities (artifacts) used as resources, along with tools to support their business activities. Accordingly, we first present the simpA-WS framework and the related simpA language – an extension of Java aimed at capturing the A&A metaphors as first-class entities; we then show how agents and artifacts can be programmed in simpA and how SOA/WS applications can be programmed in simpA-WS; a simple running example is discussed for concreteness.

I. Introduction

Nowadays Web Services (WS) represent the reference standard technologies for setting up distributed systems that need to support interoperable machine-to-machine interaction between heterogeneous applications distributed over a network [17]. In that context, Service-Oriented Architecture (SOA) appears to be more and more the reference software architecture promoted by leading industries—IBM, Microsoft, Sun, IONA, Bea, to cite few ones—as a blueprint for organising, designing and building distributed enterprise applications based WS open set standards [2], [4].

Generally speaking, SOA can be defined as an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, inter-operable, discoverable, and potentially reusable services [4]. From a software architecture perspective, SOA defines how to use loosely coupled software services for supporting the requirements of the business and software users, making resources available on a network as independent services that can be accessed without knowing their implementation platform. From an information systems perspective, SOA enables the creation of applications by combining loosely-coupled, inter-operable services. Despite the specific perspective, service-oriented architectures based on Web Services are well going to be adopted by the industry as the reference choice for inter-operable, distributed systems.

A key issue here is the *programming model* to be adopted for SOA applications [7] – that is, the model defining the concepts and abstractions made available to developers. From this viewpoint, SOA *per-se* is not committed to any specific programming model: however, the ones currently promoted by leading software vendors are essentially *component-based* [16], so as to rely on mature and widespread technologies. Yet, in this paper we argue that such a choice is unable to handle some essential requirements of SOA systems—such as autonomy, control-uncoupling and data / message-driven interactions—at a suitable abstraction level. For this reason, there is a need for a programming model based on *agent-oriented abstractions*, which makes it possible to deal with such requirements in an effective and more natural way.

Agents and Multi-Agent Systems (MAS) have already been recognised as suitable approaches for engineering complex, intelligent service-oriented applications, aimed at integrating research outcomes from different contexts such as Semantic Web and Artificial Intelligence [10]. Here, however, we explicitly focus on designing and programming issues, discussing how agents and MAS could provide effective building blocks for the design and development of SOA applications.

The remainder of the paper is organised as follows. In Section II we focus on the requirements that any programming model for SOA application programming should satisfy, and briefly review from this viewpoint the main programming models currently promoted by the industry. Then, in Section III we introduce an agent-oriented programming model for SOA/WS based on agents and artifacts. In Section V we present simpA-WS, as a simple Java-based middleware for supporting such a programming model; conclusions are drawn in Section VI.

II. Background: SOA Programming Models

In order to evaluate the effectiveness of a programming model for SOA, it is first necessary to outline the main properties that a SOA system should exhibit according to the reference literature (see for instance [2], [4]).

Encapsulation is the basic property for achieving service independency from the context: services encapsulate their logic, whose size and scope can vary, and can possibly encompass the logic provided by other services; in other words, one or more services can be composed into a collective service.

Autonomy is strongly related to encapsulation, since services must clearly have control over the logic they encapsulate. As

A. Ricci is with the DEIS Department, Università di Bologna, Cesena.
E. Denti is with the DEIS Department, Università di Bologna.

a consequence of such an autonomy, inter-service relationships should minimise dependencies – in particular, control dependencies – retaining only the *awareness* of each other: this is what we mean by *loose coupling*. Such an awareness is achieved through the use of *service descriptions*, which are exploited by users to understand how to use and interact with the service. Communication is another fundamental dimension in SOA, since services must exchange information in order to interact and accomplish their task.

Autonomy, encapsulation and loose coupling properties clearly condition the *interaction model* that can be used to enable communication both between the service user and service providers, and among services. In principle, any interaction model capable of preserving loosely coupled relationship can be adopted: *messaging* is the reference communication framework typically considered for this purpose. Conversely, interaction models based on Remote Procedure Call (RPC) or method invocation are inadequate, since they involve a *control coupling* between the interacting parts. This is indeed quite a critical aspect: most of the frameworks currently proposed as killer technologies for the rapid prototyping of Web Service applications adopt a pure OO-style in defining and interacting with Web Services, mapping – for instance – service invocations onto method invocations. A clear example of this trend is the programming model adopted by the Java API for XML Web Services (JAX-WS) [8], which defines a Web Service by simply annotating the corresponding Java class class with the @WebService annotation, and its methods – which implementing the Web Service operations – with the @WebMethod annotation. An analogous support can be found in the Web Service Extension (WSE) provided by the Microsoft .NET platform.

Our view is that the above critical aspect is mainly due to a fundamental mismatch between the SOA and object-orientated paradigm – with the object-oriented paradigm often adopted to engineer distributed (and concurrent) systems – rather than to weaknesses in today's technologies. In fact, although it is possible to build such kinds of systems on top of available OO platforms exploiting middleware such as CORBA, RMI or alike, the *abstraction level* provided is inadequate for application design and implementation, in that OO lacks abstractions to deal with loose-coupled communication, concurrency, and distribution.

Consequently, new programming models are needed for implementing SOA systems, which preserve the basic properties required from Web Services. For this purpose, some proposals have been pushed by leading industries in the state-of-the-art: Service Component Architecture (SCA) [5], for instance, is promoted by independent software vendors such as IBM, SAP, IONA, Oracle, BEA, TIBCO — to cite some. Analogous initiatives are the Windows Communication Foundation (previously called Indigo), promoted by Microsoft, and the Java Business Integration (JBI), promoted by the Java Community process [9]. All such approaches adopt a *component-based* programming model: components implement the business logic, offer their capabilities to other components, and consume functions offered by other components through suitable Service-Oriented interfaces (Figure 1 shows

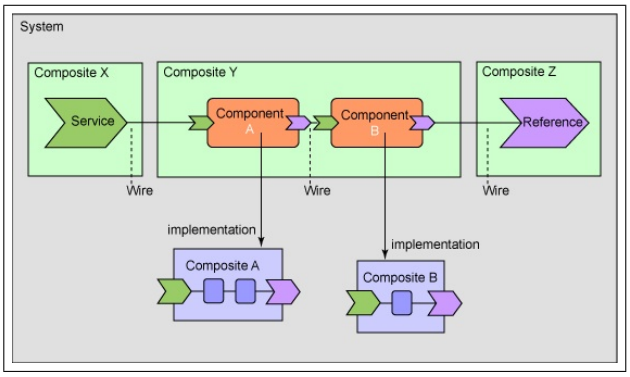

Fig. 1. An abstract representation of the Service Component Architecture, reported in [5].

abstract representation of the Service Component Architecture, taken from [5]) using a minimum of middleware APIs. Components are linked together according to some *wiring model*, which is meant to support different kinds of interaction models and features, including synchronous and asynchronous invocation, transactional behaviour of components invocation, and so on. Service implementation and service composition are uncoupled from both the details of the infrastructure and of the access methods used for service invocation: these typically include Web services, Messaging systems and CORBA IIOP.

As it can be expected, such an approach inherits on the one side the well-known strong points of the component-oriented paradigm in terms of dynamic configurability, reusability, etc., but also its weakness in dealing with processes and activities, concurrency, autonomy, distribution, decentralisation and encapsulation of control – to cite some. Neither object-oriented, nor component-based programming models provide first-class abstractions to explicitly model and manage the above issues: in particular, both objects and components are passive entities encapsulating their state and behaviour, but not the control of such a behaviour, which is typically hidden in some part of the component's "container" – whatever this may be. As a consequence, even if components are meant in principle to encapsulate the business-level logic, they fail to encapsulate some key aspects of such a logic – such as, for instance, the execution and control of (possibly concurrent, possibly interacting) business activities and processes. To overcome these limitations, in the next section we introduce a programming model based on *agent-oriented* abstractions, aimed at capturing the above aspects in a full-fledged way. ***troppo forte?***

III. An Agent-Oriented Programming Model for SOA and Web Services

Interestingly, the word *agent* appears both in the abstract description of the Web Service reference architecture provided by W3C [17] (sketched in Figure 2), and – more generally – in the high level characterisation of SOA [4]. There, an agent is used to represent:

- the *service requestor*, which encapsulates the business logic on how to *use* services: from an interaction point of view, this results in sending and receiving messages in compliance with the service interface specification;

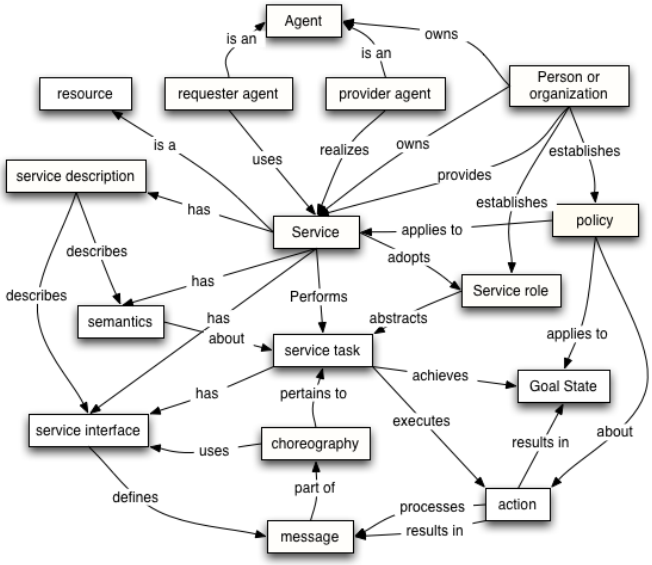

Fig. 2. Service Model of Web Services, according to W3C

- the *service provider*, which encapsulates the business logic of the service: this processes the requestor messages, executes the related activities and interacts with the requestor via the message exchange protocol specified in the service description.

So, some notion of agent already appears in the standards as a key part of the picture, representing the entities that perform some activity or achieve some goal, thus shaping the business logic either on the user's or on the service's side. However, such abstraction level disappears when moving from the abstract characterisation down to the design and development levels, as discussed in the previous section. Our proposal is to *keep that abstraction level alive* throughout the engineering process, exploiting agents and MAS as the basic bricks of a programming model explicitly tailored to the definition of services and of applications using such services.

The fundamental outcome of this approach is to reduce the gap between the business-level description and the models and architectures used at the system implementation level.

In fact, despite the differences between the existing agent-oriented methodologies, models and architectures, the agent-oriented paradigm *in se* provides precisely the high-level concepts — activity, goal, task, message-driven interaction, ...— that are needed from a programming model in order to map the metaphors used at the business description level. In the next Section we introduce an agent-oriented programming model called SA&A, based on a the A&A conceptual model.

A. The A&A Conceptual Model

A wide range of agent programming models, architectures and platforms can be found in literature (see [6] for a brief survey of the programming languages and platforms). For historical reasons, most of them are AI-oriented, thus with a characterisation of the agent and MAS abstractions focussed on AI concepts, aimed at building systems exhibiting a somewhat intelligent behaviour. The *Agents and Artifacts* conceptual model (A&A henceforth) [12], instead, was defined with a software engineering perspective in mind: as such, it highlights the features needed for an effective design and development of complex software systems.

Grown from inter-disciplinary studies involving Activity Theory and Distributed Cognition [11], A&A adopts *agents* and *artifacts* as high-level abstractions to design and build distributed, concurrent software systems. These metaphors are taken from human cooperative working environments, where "systems" are composed by individual autonomous entities (humans) who pro-actively carry on some kind of work (activities) by interacting and cooperating. A fundamental aspect of such cooperative systems is the context—i.e. the *environment*—that makes it possible for such activities to take place. Humans cooperative environments are full of suitable artifacts and tools, that humans produce, consume and use to support their work. Following Activity Theory, the term *artifact* is used here to identify both the resources and objects constructed during the activities, as well as whatever instrument built or exploited by humans to support their activities.

In A&A these metaphors are brought into the software engineering process, modelling complex software systems in terms of *workspaces* where ensembles of pro-active entities—the agents—work together by producing, consuming, sharing and cooperatively using different kinds of artifacts, analogously to the human case.

B. Agents and Artifacts

Agents represent entities with a (pro-)active behaviour, designed by engineers so as to perform some kind of useful work, cooperatively and concurrently to the work of the other agents. The agent abstraction is well suited for encapsulating the execution and control of the business activities and processes that are part of the business logic. Artifacts, in turn, represent passive entities that populate the agents' working environment: they are designed by engineers as resources and tools to be used by agents for their (individual or collective) work.

So, on the agent side, A&A promotes an *activity-oriented* model, where the agents' pro-active behaviour is modelled in terms of activities whose execution and control is fully encapsulated inside the agent; on the other, agents manipulate, produce, exploit, update artifacts which constitute the tools needed for their work.

Activities are expressed in terms of *actions*, that is atomic step determining some kind of change either in the agent state (internal actions) or in the environment (external actions or simply actions). *Sensing*—representing here the action of perceiving —is the basic mechanisms that enables an agent to get information from its environment.

Artifacts are used by agents as source or target of their work, and greatly vary nature and function — including, for instance, the tools for enabling agent communication and coordination such as blackboards, message boxes, and calendars, which are typical *coordination artifacts* [13]. Instead, shared knowledge bases or artifacts representing or wrapping I/O devices are typical examples of *resource artifacts*. Each artifact is explicitly designed by MAS engineers to encapsulate some kind of

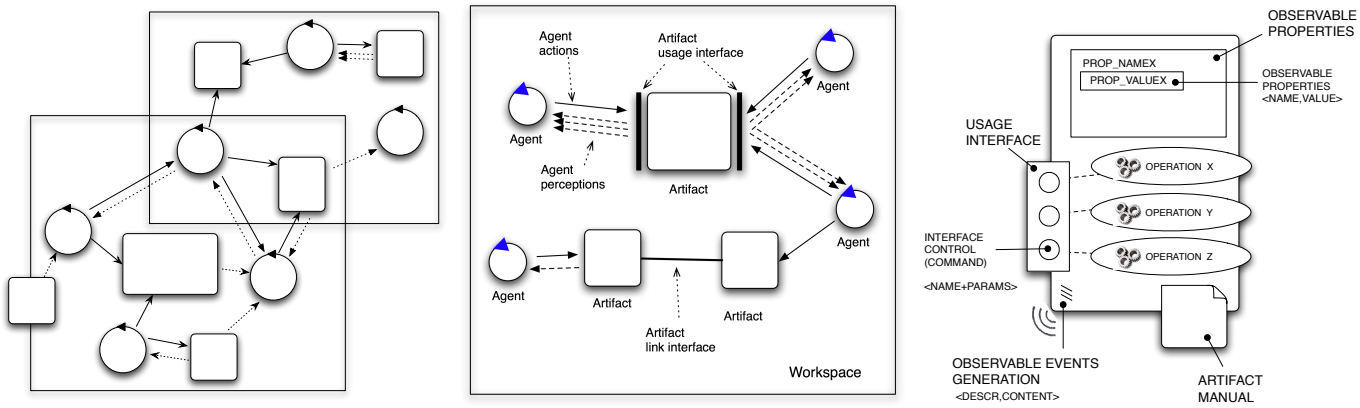

Fig. 3. *(Left)* An abstract representation of an application according to the A&A programming model, as a collection of agents (circles) sharing and using artifacts (squares), grouped in workspaces. *(Center)* An abstract representation of an agent, as an entity executing actions and getting perceptions from the environment where it is logically situated. *(Right)* An abstract representation of an artifact, with its usage interface and observable properties in evidence.

function, here synonym of "intended purpose"; any function is structured into a set of *operations*. In order to be used by agents, each artifact exposes a *usage interface*, which defines the set of controls on which the agents can act upon so as to trigger and control the execution of operations. Such execution can result in the generation by the artifact of *observable events*, that can be perceived by the agents which are using the artifact. Usage interface controls have a name and possibly parameters, which must be specified by agents when using the artifact.

Summing up, the interaction between agents and artifacts is based on the notions of *use* and *observation*, and strictly mimics the way in which humans use their artifacts. As a simple example, a coffee machine is an artifact whose usage interface provides controls to make coffee and select the sugar level, and whose state and behaviour are observable by the generation of events exposed through a display.

Artifacts can also be composed together by means of *link interfaces*, which make it possible to create complex artifacts as dynamic compositions of existing simpler artifacts.

Although a detailed description of A&A is outside the scope of this paper (interested readers are referred to [12]), our aim here is to identify some essential properties that make it an interesting reference for a SOA programming model. First, the agent abstraction explicitly enables and captures the *encapsulation* of control, along with a notion of *autonomy* as depicted by SOA requirements. Moreover, the interaction model adopted for agents and artifacts interaction is strongly uncoupled and data-oriented (vs. control oriented), thus providing for *uncoupled and data-driven interaction*: in fact, there are no flows of control from an agent to an artifact or other agents, as it happens instead in the case of Remote Procedure Calls (RPC) or classical object-oriented method invocation. Finally, *concurrency* can be naturally modelled both in the form of concurrent activities carried on by an individual agent, and as separate works carried on independently by distinct agents (*seamless concurrency support*).

C. A SOA/WS Programming Model Based on A&A

In this section we introduce a basic programming model for SOA based on A&A abstractions, referenced in the following as SA&A. Both services and service-user applications in SA&A are uniformly modelled as a workspace where an ensemble of agents work together, interacting both via direct communication and by producing, consuming, sharing and cooperatively using a dynamic set of artifacts. Agents encapsulate the responsibility of the execution and control of the business activities that characterise the SOA specific scenario, while artifacts encapsulate the business resources and tools needed by agents to operate in the application domain. Figure 4 represents an abstract picture of a (web) service designed upon the SA&A programming model: agents act as service providers processing incoming service messages, but some of them also act as clients of other services, as an example of service composition.

Two kinds of artifacts are used in almost any service-oriented application: ws-service-panel and ws-service-interface. Both are used as *interfaces* or *media* enabling the communication with the service clients or with external Web services, based on open standards. In particular, the former is used by agents implementing the business logic to retrieve and be aware of the requests and messages sent to the (web) service: so, for each service, one instance of such an artifact encapsulates the functionalities related to a specific WSDL and WS-Policy service description. In the simplest model, the usage interface of this artifact provides just controls to manage messages and requests—for instance, a control to retrieve the messages to be processed, another to send response messages, one further to check the number of pending messages, etc. In more complex models, however, this artifact could encapsulate the management of some quality-of-service aspects (such as security, reliability, etc.), as defined by WS-* specification. The latter one, instead, is used by agents to interact with an existing service. So, an instance of the ws-service-interface is usually first instantiated referring to a specific WSDL and possibly WS-Policy description, and then used (by one or more agents) to interact with a specific Web Service. In the simpler case, its usage interface should provide controls just to invoke services and observe possible response messages. However, other functionalities could be encapsulated here for the management of QoS aspects (described in the WS-Policy

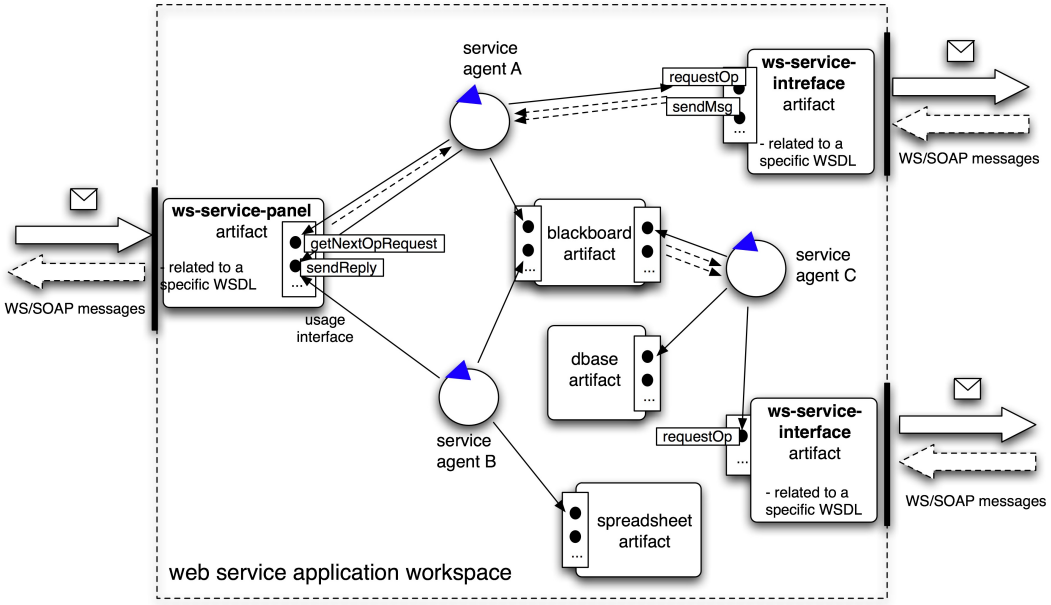

Fig. 4. An abstract representation of a (web) service architecture according to the SA&A programming model, composed by agents and artifacts building blocks. Artifact usage interface is represented as a panel with some controls inside. Some of them are labelled with a name which is equal to the operation that the control is meant to trigger. The ws-service-panel and ws-service-interface artifacts in the figure are used respectively to collect request messages and to interact with existing (web) services.

specification).

In the abstract representation in Figure 4, other kinds of general purpose artifacts are represented, such as a shared knowledge base, a blackboard, a spreadsheet. Specific kind of artifacts could be instantiated dynamically, or disposed of, according to the evolution of the service provision. Two remarks are worth before closing this section. First, the picture refers to the service side of a SOA: the client-side would be similar, yet with no the need for ws-service-panel artifacts[1] Moreover, only the basic aspects of a service have been presented, since our aim is to give the reader the "taste" of the shift from state-of-the-art, component-based to agent-oriented approaches, rather than developing a full-fledged application scenario.

IV. Programming Agents and Artifacts in simpA

simpA is an open-source extension[2] of the Java platform aimed at assuming the A&A abstractions as the basic high-level building blocks to program concurrent applications [15]. This approach contrasts with most of the current approaches, which often model concurrency aspects by "adapting" object-oriented abstractions (classes, objects, methods)—e.g. [3]. Rather, we introduce the new A&A abstractions, and exploit real object-orientation to model any basic low-level data structure used to program agents and artifacts, as well as any information exchanged through interaction. This approach leaves concurrency and high-level organisation aspects orthogonal to the object-oriented abstraction layer, leading, in principle, to a more coherent programming framework.

Currently, the simpA extension is realised as a library, exploiting Java annotations to define the new programming constructs: consequently, a simpA program can be compiled and executed using the standard Java compiler and virtual machine, with no need for specific extensions of the Java framework (preprocessors, compilers, class loaders, or JVM patches). Hence, the newest constructs take the form of annotated classes and methods—which, however, are clearly separated from their non-annotated, underlying object-oriented versions used at the implementation level. The choice of using the library & annotations solution to implement a language and a platform extension has the advantage to maximise the reuse of a widely adopted platform like Java: at the same time, it has some relevant drawbacks, due to the lack of agents and artifacts as first-class abstractions both in the language and in the virtual machine. Accordingly, part of our ongoing work is devoted to the definition and the prototype implementation of a new full-fledged language and platform called simpAL, rooted on agents and artifacts as real first-class entities.

In the remainder of the section we first describe how to define the structure of an agent (Subsection IV-A) and of an artifact (Subsection IV-B), then present the API supporting the agent-artifact interaction (Subsection IV-C) and the overall shape of a simpA application (Subsection IV-D).

A. Defining Agents

Since one of our main objectives was to minimise the number of classes to be defined by users for introducing new agents and artifacts, we adopted a very simple, one-to-one mapping—just one class per agent or artifact template—so as

[1]Of course, agents would encapsulate the business activities of the client side of the application.

[2]The simpA technology is available for download at the simpA web site, http://www.alice.unibo.it/simpa

to make things as agile as possible. Accordingly, a new agent template[3] is defined by extending the `alice.simpa.Agent` base class provided in the simpA API: the class name is equal to the agent template's name. At runtime, new instances of this agent type can then be spawn when needed. The execution of an agent consists in executing the activities specified in its template, starting from the main one.

In the following, we stress the four key aspects of agent templates' definition: the *memo-space* as a way to provide long-term memory (Subsection IV-A.1), the definition of *atomic vs. structured activities* (Subsection IV-A.2), the *coordination* of such (sub-)activities (Subsection IV-A.3), and the definition of *cyclic behaviours* (Subsection IV-A.4).

1) Agents' long-term memory: the memo space: 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 *memos*. A memo is a tuple, characterised by a label and an ordered set of arguments, possibly bound to data objects. If some arguments are left unbound, the memo is partially specified. A *memo-space* is just a dynamic set of memos: each memo is identified by its label and argument list.

Each agent is provided of internal actions—available in the implementation as protected methods—to atomically and associatively access and manipulate the memo space. In particular, `memo(`*`Label`*`,`*`Arg0`*`,`*`Arg1`*`,...)` is used to create a new memo with a specific label and arguments: these can be null or bound to specific data objects. Conversely, `readMemo(`*`Label`*`,`*`Arg0`*`,`*`Arg1`*`,...):Memo` and `removeMemo(`*`Label`*`,`*`Arg0`*`,`*`Arg1`*`, ...):Memo` respectively read and remove a memo that matches both the label and the given arguments: these can be either concrete values or variables—in the latter case, represented as instances of the `MemoVar` class. In the special but frequent case that, due to the designer's own choice and convention, the label alone is enough to uniquely identify the memo type—that is, the same label in not used twice with a different argument list to denote different memo types—and that a single tuple of a given type is present in the memo space at a time, two linguistic shortcuts are provided: `getMemo(`*`Label`*`):Memo` and `delMemo(`*`Label`*`):Memo` respectively get and remove a memo with a given label, chosen non-deterministically among the existing ones.

By default, the `boot_args(`*`Arg0`*`,`*`Arg1`*`,...)` memo is available in each agent's memo space at the agent's boot time, and contains the parameters optionally specified when the agent has been instantiated.

It is worth remarking that the memo-space is the only data structure adopted for supporting the agent's long-term memory: the instance fields of agent classes are not used.

2) Atomic and structured activities: Agent activities can be either *atomic*—i.e. not composed of sub-activities—or *structured*, composed by some kinds of sub-activities. 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 behaviour 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 agent template. Here is a naïve example of agent template:

```
public class MyAgent extends Agent {
  @ACTIVITY void main(){
    log("Hello, world!");
  }
}
```

In this case, the agent behaviour simply logs the "Hello, world" message onto standard output and then terminates.

Structured activities are (hierarchically) composed of 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 sub-activity to be executed, and optionally a pre-condition. When a structured activity is executed, all the *todo*s in the agenda are executed as soon as their pre-conditions hold: no pre-condition means that the *todo* can be executed immediately. So, multiple sub-activities can be executed concurrently in the context of the same (super) activity.

A structured activity is implemented by a method annotated with an `@ACTIVITY_WITH_AGENDA` annotation, which contains the *todo* descriptions as a list of `@TODO` annotations. Each `@TODO` specifies the name of the sub-activity to be executed, as well as a `pre` property for the optionally precondition, expressed as a boolean expression of Prolog predicates, possibly combined through the classical *and*, *or* and *not* connectors (represented by the `,`, `;`, and `!` symbols, respectively). Predicates can be predefined or user-defined—actually, any valid Prolog expression (clause body) can be specified. Essentially, these predicates make it possible to specify conditions on the current state of the activity agenda, in particular on *(i)* the state of the sub-activities (*todo*s)—whether they have completed / aborted / started, and on *(ii)* the *memos* that could have been attached to the agenda. Preconditions can depend only on the local (inner) agent's state, not on the agent-environment state.

Now let us see a simple example of an agent with a structured activity, whose agenda is composed by four *todo*s: `activityA`, `activityB`, `activityC`, and `activityD`

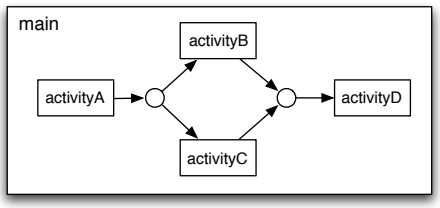

Fig. 5. A representation of an agent's main structured activity composed of two parallel sub-activities activityB and activityC to be executed after activityA; activityD is executed after the completion of both activityB and activityC.

[3]The term "template" is used here as a higher-level synonym of "class", intended as the entity describing the structure and behaviour of all the template instances.

(see Figure 5). `activityA` is meant to be executed as soon as the main activity starts, `activityB` and `activityC` are executed in parallel when `activityA` completes, while `activityD` starts when both `activityB` and `activityC` have been completed.

```
public class MyAgent extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO("activityA"),
    @TODO("activityB", pre="completed(activityA)"),
    @TODO("activityC", pre="completed(activityA)"),
    @TODO("activityD",
       pre="completed(activityB),completed(activityC)")
  }) void main(){}

  @ACTIVITY void activityA(){
    memo("x",1);          // attach a new memo x(1)
  }

  @ACTIVITY void activityB(){
    int v = getMemo("x").intValue(0); // retrieve 1st arg
    memo("y", v+1, v-1);  // attach a new memo y(2,0)
  }

  @ACTIVITY void activityC(){
    memo("z", getMemo("x").intValue(0)*5); // attach z(5)
  }

  @ACTIVITY void activityD(){
    MemoVar y0 = new MemoVar();
    MemoVar y1 = new MemoVar();
    readMemo("y",y0,y1);  // read memo arguments
    int z = getMemo("z").intValue(0);          // z =  5
    int w = z*(y0.intValue() + y1.intValue());  // w = 10
    log("the result is: "+w);            // should log 10
  }
}
```

In this example, the agent attaches and retrieves some memos in the memo-space to share data among its (sub-)activities and store the result of its work. In particular, in `activityA` the agent stores a memo `x(1)`, then in `activityB` and `activityC` reads the memo labelled with `x` and uses its content to create the two new memos `y(2,0)` and `z(5)`; finally, in `activityD`, it reads both memos `y` and `z` and uses them to compute the desired result. The `Memo` class provides methods for accessing the memo content: for instance, `intValue(`*i*`)` retrieves the *i*-th argument as an integer value.

It is worth noting that local method variables are exploited as a kind of *short-term memory*, in contrast with the memo-space exploited as a long-term memory.

3) Coordinating sub-activities: Memos can be used both to contain data objects elaborated by activities, and to support the coordination of sub-activities. This is possible by exploiting the `memo` predicate in the specification of the pre-conditions so as to test the presence of a specific memo in the memo space—and possibly to associatively retrieve its argument values, if needed. Below is a variant of the previous example, where the pre-conditions for the execution of sub-activities are no longer expressed as conditions on the completion of other activities, but are based on the availability of the information that each sub-activity needs in order to be executed:

```
public class MyAgent extends Agent {
  @ACTIVITY_WITH_AGENDA({
    @TODO("activityA"),
    @TODO("activityB", pre="memo(x(_))"),
    @TODO("activityC", pre="memo(x(_))"),
    @TODO("activityD", pre="memo(y(_)),memo(z(_))")
  }) void main(){}
  ...
}
```

Accordingly, `activityB` is triggered as soon as a memo matching the template `x(_)`[4] is found in the memo space, and the same for the other activities.

4) Cyclic behaviour: In order to define a cyclic behaviour, a *todo* can be specified to be *persistent*: then, once it has been completely executed, it is automatically re-inserted in the agenda, so that it is eventually executed again. In the following example, the agent's main activity consists of repeatedly acquiring a new task to do and serving it concurrently with the other running tasks.

```
public class MyAgent extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO("getNewTaskTodo", persistent=true),
    @TODO("doTask", pre="memo(new_task_todo)",
                    persistent=true)
  }) void main(){}

  @ACTIVITY void getNewTaskTodo(){
    // <wait for a task todo>
    memo("new_task_todo");
  }

  @ACTIVITY void doTask(){
    removeMemo("new_task_todo");
    // <do task>
  }
}
```

B. Defining Artifacts

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

In particular, for each operation (control) listed in the usage interface, a method with no return parameter and annotated with the `@OPERATION` annotation must be defined: the method name and arguments must coincide with the name and arguments of the operation to be triggered. Any method annotated with `@OPERATION` represents the first computational step executed when the homonymous operation is triggered. Moreover, since any useful artifact has to be somehow *observable*, the `signal` primitive is used to generate *events* that can be observed by the agent using the artifact.

As a simple example, the following code shows the definition of a `Count` artifact functioning as a simple counter, whose usage interface defines just one operation (`inc`) for incrementing the counter value:

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

 public Count(){ count = 0; }

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

[4]Following the Prolog syntax, the underscore means *any value*. Analogously, symbols starting with an uppercase letter represent variables.

An observable event is characterised by a label describing the kind of the event and possibly an object representing the event data. In the previous example, for instance, a `new_count_value` event is generated each time the counter is updated.

Some events are automatically generated for any operation execution: in particular, `op_execution_completed` and `op_execution_failed` are generated when an operation completes with a successful or a failure result, respectively.

Besides observable events, an artifact can define a number of *observable properties*—that is, labelled inner state variables whose change is made observable to agents which are *focussing* the artifact (this aspect is discussed more in detail in Subsection **??**). Observable properties are expressed as instance fields annotated with the `@OBSPROPERTY` annotation: a basic set of primitives is available to manipulate the property values. As an example, let us consider a variant of the `Count` artifact, which defines the `count` observable property:

```
public class Count extends Artifact {
  OBSPROPERTY int count;

  public Count(){ count = 0; }

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

Now, each time the operation `inc` is executed, the property value is updated by the `updateProperty` primitive, which causes the generation of an observable event of type `property_updated(count)`: the event data carry the new property value.

In the following, we stress more in detail three key aspects of artifact definition: the definition of structured operations (Subsection IV-B.1), temporal guards (Subsection IV-B.2), and linkability (Subsection IV-B.3); other artifact features are reported in Subsections IV-B.3, IV-B.4 and IV-B.5.

1) Structured operations: In previous examples, artifact operations were always atomic—i.e., made of a single step. However, structured operations, composed of multiple (atomic) steps, can also be implemented: to this end, each operation step has to be encoded by a method annotated with `@OPSTEP` annotation, which is triggered by the `nextStep` primitive. This primitive specifies the name of the step to be triggered along with its parameters, as a kind of continuation.

In addition, each operation or operation step can be provided with a *guard*, that is, a condition that must hold for actually executing the triggered operation or operation step. Guards are implemented as boolean methods annotated with the `@GUARD` annotation, whose arguments must exactly match those of the guarded operation or operation step: as soon as the guard is evaluated to true, the step is executed. For the sake of concreteness, let us consider the following example:

```
public class MyArtifact extends Artifact {
  int m;

  @OPERATION void op1(){
    m = 1;
    nextStep("opStepB");
  }

  @OPSTEP(guard="canExecOpStepB") void opStepB(){
    log("op1 completed.");
  }

  @GUARD boolean canExecOpStepB(){ return m == 5; }

  @OPERATION void op2(){ m++; }
}
```

Here the operation `op2` is atomic, while the operation `op1` is composed of two steps: the first coincides with the operation itself[5] (and initialises `m` to 1), while the second, `opStepB`, is explicitly labelled and encoded by the homonymous method (which writes a message to the log). Of course, the definition order of these methods is not significant—the above writing order is just a matter of readability.

The `opStepB` step is triggered by the first step in `op1` through the explicit invocation of the `nextStep` primitive: once triggered, the step is executed only when (and as soon as) its guard, `canExecOpStepB`, evaluates to true. This guard conditions the step execution to the value of the internal artifact variable `m`, which must be equal to 5 for `opStepB` to be actually executed. In turn, `m` is incremented by the operation `op2`: so, `opStepB` is executed only after agents have invoked `op2` four times, raising `m` to 5. This completes the execution of operation `op1`.

As anticipated above, a guard can also be applied to an operation, with the obvious meaning that the guard must be true for (the first step of) the operation to be executed:

```
public class MyArtifact extends Artifact {
  int m;
  public MyArtifact(){ m=1; }
  @OPERATION(guard = "canExecOp1") void op1(){ m++; }
  @GUARD boolean canExecOp1(){ return m < 10; }
}
```

It is worth noting that multiple instances of the same operation can be triggered for execution, typically by distinct agents: in that case, a strict ordering semantics applies on their execution, based on to the time when the operation has been triggered (besides the evaluation of the guard).

Summing up, structured operations make the implementation of long-term operations encapsulated, flexible and effective, allowing for multiple structured operations to be executed concurrently by interleaving the guarded execution of their steps, while enforcing the mutual exclusion on the access to the artifact state. For further details, we forward the interested reader to the simpA manual available at [1].

2) Temporal guards: Besides guards based on the artifact state, simple *temporal guards* are also supported: their evaluation becomes true after a given time is elapsed since they have been triggered. To define a temporal guard, a `tguard` property must be specified inside the `@OPSTEP` annotation instead of `guard`: the property should then be assigned a (long) positive value, indicating the guard duration in milliseconds. Let us consider the following example:

```
public class Clock extends Artifact {
  OBSPROPERTY int nticks;
  boolean stopped;

  public Clock(){ nticks = 0; stopped = false; }
```

[5]This choice makes it easier to express single-step operations, which are the most frequent case.

```
  @OPERATION void start(){
    stopped = false;
    nextStep("tick");
  }

  @OPSTEP(tguard=1000) void tick(){
    if (!stopped){
      updateProperty("nticks",nticks+1);
      nextStep("tick");
    }
  }

  @OPERATION void stop(){ stopped = true; }
}
```

Once started via the `start` interface command, this artifact generates a tick event approximatively every second, which increments the clock value. When some agent issues the `stop` command, the counting finally terminates.

3) Linkability: Besides the usage interface, artifacts can be provided with a *link interface*, that is, a set of operations to be invoked (*linked*) by other artifacts (not by agents). This feature makes it possible to create complex artifacts by dynamically composing simpler artifacts, mimicking the way in which human artifacts are linked together. From the concurrency model viewpoint, linked operations have the same behaviour of operations triggered by the usage interface: the only difference is that the observable events generated by a linked operation are made observable not to the artifact linking the operation (which would not make sense in the simpA model), but to the agent originating the execution chain.

4) Observable states: The observable behaviour of an artifact can be partitioned in states, each equipped with a different usage interface and observable property set.

5) Artifact manual: Finally, each artifact should be equipped with a document containing a formal machine-readable, semantic-based description of the artifact functionality and usage instructions (usually called operating instructions). Such a description is meant to provide a semantic layer making it possible to envision, in principle, scenarios where agents could be able to select and use artifacts that are added dynamically to their working environment, without a pre-programmed knowledge about their functionality and use. Again, the interested reader is forwarded to the simpA web site for up-to-date documentation on the work in progress.

C. Agent API for Interacting with Artifacts

The API for enabling agents to interact with artifacts concerns two main categories: use and observation (Subsection IV-C.1) on the one side, instantiation and discovery (Subsection IV-C.2), on the other.

1) Artifact use and observation: Artifact *use* is the basic form of interaction between agents and artifacts. In fact, also artifact instantiation and artifact discovery are realised by using proper artifacts—a *factory* and a *registry* artifacts—which are supposed to be available in any working environment.

Following the A&A model, the use of an artifact by an agent involves two basic aspects: *(i)* executing operations on the artifact, and *(ii)* perceiving the observable events possibly generated by the artifact through agent sensors.

Agents execute operations on artifacts by exploiting the interface controls (or commands) provided by the artifact usage interface. The `use` basic action is provided for this purpose, and specifies 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 value returned by `use` is the operation's unique identifier. If, instead, the action execution fails—for instance, because the interface control specified is not part of artifact usage interface—an exception is raised.

Sensors are represented by specific classes, which extend the basic abstract class `alice.simpa.AbstractSensor`. A concrete default implementation is provided, called `alice.simpa.DefaultSensor`. Default sensors provide a simple FIFO policy in managing observable events collected from the environment. Sensors are dynamically created as normal objects—specifying a logic name—and are then dynamically linked to / unlinked from agents' bodies. An agent can link / unlink any number of sensors, possibly of different kind, according to its own strategy for sensing and observing the environment, by means of specific primitives (`linkSensor`, `unlinkSensor`, and `linkDefaultSensor`).

The `sense` primitive makes it possible to retrieve the events collected by a sensor: it waits until a matching event appears — that is, until an event collected by a given sensor matches an optional pattern (for data-driven sensing) — or until an optional timeout is reached. Pattern matching is based on regular-expression patterns, matched over the event type (a string). In the case of a successful execution, 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, instead, no perception is sensed for specified time, a `NoPerceptionAvailableException` is raised. The following code shows the `CountUser` agent creating and using a `Count` artifact, then locating and using the `myArchive` artifact (instantiation and discovery will be described later).

```
public class CountUser extends Agent {
  @ACTIVITY void main() {

    SensorId sid = linkDefaultSensor();
    ArtifactId countId = makeArtifact("myCount","Count");

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

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

      ArtifactId dbaseId = lookupArtifact("myArchive");
      use(dbaseId, new Op("write",new DBRecord(value));

    } catch (NoPerceptionException ex){
      log("No count_value perception from the count");
    }
  }
}
```

The agent activity accounts for: *(i)* creating a `Count` artifact as described in previous section; *(ii)* using the artifact, executing twice the `inc` operation provided by `Count` usage interface and observing the `count_event` generated by the artifact (carrying the count value) only the second time that

the operation is executed; *(iii)* locating and using a `DBase` artifact called `myArchive`, performing the `write` operation to record the value perceived by `myCount`.

The class `ArtifactId` is exploited to represent artifacts' unique identifiers. The first time that an agent executes `inc`, it is not interested in observing the events generated by the operation execution, so no sensor is specified. The `sense` primitive is used to perceive only the events matching `count_value`, and with a timeout of one second. The `Perception` class provides a `getContent` method to get the content of the perception (event).

It is worth remarking here the similarities but also the deep differences between the notion of sensor, on the one side, and the *future* construct / pattern used in concurrent programming for handling asynchronous calls, on the other. In fact, a sensor can be used to collect possibly-multiple observable events generated as a consequence of possibly-multiple `use` actions, on possibly-distinct artifacts. Futures, instead, are typically used to get asynchronously the single result of the execution of a single call.

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 starts observing the artifact, and therefore specifies the sensor to be used to collect the events and optionally the reg-ex filter defining the events to be observed; the latter obviously stops the observation process.

2) Artifact instantiation and discovery: In the above example two further actions, `makeArtifact` and `lookupArtifact`, are used to instantiate and lookup artifacts. As briefly mentioned in the previous Subsection, both these auxiliary actions are realised on top of a bunch of `use` and `sense` actions executed on two pre-defined artifacts available in each simpA application—the `factory` and the `registry` artifacts, respectively.

In particular, instantiation is actually handled by the factory's `makeArtifact` operation, which takes the logical name of the new artifact, its template (full class name or `Class` type) and possibly the parameters needed for its creation (which coincide with the constructor parameters of the artifact template class). In case of success, a `make_succeeded` event is generated, and the artifact identifier is provided as the event content. The factory also provides an analogous operation, `makeAgent`, to instantiate & spawn agents.

In quite the same way, artifact discovery is handled by the registry's `lookupArtifact` operation, whose argument is the name of the artifact to be located: upon success, the artifact identifier is returned as the content of a `lookup_succeeded` event.

D. A simpA Application

The core of a simpA application is a simple main class where a simpA working environment is created and an initial sets of agents are booted, possibly with a starting set of artifacts.

In the following example, the `my-app` workspace is created, initially composed of an instance of the `DBase` artifact (not reported) and a `CountUser` agent (which in turn will create a `Count` artifact):

```
public class MyApp {
 public static void main(String[] arg) throws Exception{
   ISimpaWorkspace env = Simpa.createWorkspace("my-app");
   env.createArtifact("myArchive","DBase");
   env.spawnAgent("a-user","CountUser");
 }
}
```

Any non-naïve application, however, can be expected to demand the creation of multiple agents working concurrently in a workspace populated by multiple artifacts of different kinds.

V. Programming SOA/WS Applications on Top of simpA: The simpA-WS Framework

simpA-WS is a framework on top of simpA which makes it possible to build Web Service applications as simpA workspaces with agents and artifacts, following the model described in Subsection III-C. Besides simpA, simpA-WS exploits Apache AXIS2 open-source technology[6] as an enabling low-level Java-based web service technology effective for managing (SOAP) message exchange, for an effective management of XML data, etc. simpA-WS is a fully Java-based open-source technology, and can be downloaded at the simpA-WS web site[7].

A. simpA-WS Programming Interface

The simpA-WS framework provides a uniform model to conceive both Web services and applications interacting with Web Services as simpA workspaces with agents and artifacts encapsulating the business logic: such pre-defined artifacts can be exploited as part of the service-oriented infrastructure. There are two basic kinds of artifacts:

- `ServiceInterface`—an implementation of the ws-service-interface artifact, meant to be instantiated and used by agents to interact with a specific Web Service, to send messages for executing operations and to get the replies sent back by the service. Multiple `ServiceInterface` artifacts can be instantiated and used in a given workspace to interact with multiple Web services.
- `ServicePanel`—an implementation of the ws-service-panel artifact, meant to be used by agents in a service application to manage the incoming requests and messages from the Web Service. One `ServicePanel` must be created and used for each (web) service to implement. Multiple `ServicePanel` artifacts can be instantiated and used in the same workspace to implement multiple services within the same simpA-WS application.

So, in simpA-WS the interaction with and among Web service applications is totally conceived in terms of message exchanges; operations represent, from this point of view, the context in which message exchange protocols or MEPs take place,

[6]Apache AXIS2 web site is available at: http://ws.apache.org/axis2/

[7]http://www.alice.unibo.it/simpa-ws

as described in the service documents (WSDL, choreography, etc.). Even if this is quite obvious by considering the reference SOA/WS model, most of the existing platforms supporting SOA/WS hides the message level to the programmer, providing API where the execution of an operation typically accounts for invoking a method in stub objects or components providing an interface that mirrors the service interface. Conversely, in simpA-WS interacting with a service accounts (from an agent viewpoint) for using a `ServiceInterface` artifact to send messages to the service according to the protocol characterising by the operation to be executed.

The message level is adopted here also at the service side, while in non-agent oriented platforms the message processing is typically not realised by the components that encapsulate the service's business logic. Conversely, in simpA-WS the message level is brought to the business logic, so that one or multiple agents—encapsulating the business logic activities—exploit a `ServicePanel` artifact to retrieve and process messages, possibly sending one or multiple replies during their activities (which can be long-term).

Summing up, a Web service application in simpA-WS (both on the service and on the user's side) is programmed as a workspace where one or multiple agents create and use one or multiple `ServiceInterface` artifacts to interact, even concurrently, with services; at the same time, they share and use other kinds of artifacts that represent useful resources and tools needed to support their (cooperative) activities. On the service side, one or multiple `ServicePanel` artifacts—the latter case concerns multiple services implemented in the same Web service application—are created and used by agents to process the requests, possibly in parallel.

In the following we describe the `ServiceInterface` and `ServicePanel` artifacts more in detail, and provide a simple running example.

B. Interacting with existing Web Services

`ServiceInterface` defines a simple usage interface for executing operations on a Web Service, providing a direct support for exchanging messages realising any possible MEP, from simple ones with at-most one message input and one message output to more articulated ones—as defined by WSDL 2.0—possibly including multiple input and output messages in the context of the same operation.

A `ServiceInterface` artifact must be instantiated by specifying—as artifact configuration parameters—the URI of the WSDL containing the service description, the name of the specific service and port type (interface) "pointed" by the artifact, and a local endpoint name, which represents the endpoint to which the artifact is bound to receive messages (such as replies). Thus, the artifact usage interface provides two basic operations:

- `sendMsg`, to send a message to the service, in the context of an operation. Abstract description:
  ```
  sendMsg(opName:String, {,msgName:String,}
        msgContent:OMElement {,relatedToMsgID:String})
  ```
  where `msgName` identifies the message to be sent (according to the WSDL), `opName` is the name of the operation, `msgContent` is the content of the message (a XML piece of data, according to the XML schema described in the WSDL), and optionally `relatedToMsgID` is the identifier of the message to which this message is related to. `msgName` is optional: if it is not specified, its value is determined by accessing the WSDL. The execution of the message generates an observable event `msg_sent(msgID:String)` if it succeeded, containing the identifier of the message sent, or an event `msg_send_failed` if it fails.
- `getReply`, to get the reply to a message previously sent during an operation. Abstract description:
  ```
  getReply(msgID:String})
  ```
  where `msgID` is the identifier of the message to which the reply message must be related. When an output message related to `msgID` is received by the artifact, the operation generates an observable event `msg_reply(msg:MsgInfo)`.

Besides these two basic operations, other auxiliary operations are provided to directly support basic MEPs; for instance:

- `requestOp`, which implements the basic request-response (in-out) MEP by sending a request message and generating an event with the response message as soon as it arrives. Abstract signature:
  ```
  requestOp(opName:String, msgContent:OMElement)
  ```
  where `opName` is the name of the operation and `msgContent` is the content of the message (a XML piece of data, according to the XML schema described in the WSDL). It's worth noting that in this case the message name is automatically retrieved from the description of the operation in the WSDL. The main observable events generated by the operation are the following: `request_sent(msgID:String)` if the request message is sent successfully (`request_failed` otherwise), and `msg_reply(msg:MsgInfo)` as soon as the reply is received.

C. Processing requests and messages for a Web Service

A `ServicePanel` artifact is used to manage and control the messages which arrive to a service, providing basic functionalities to retrieve them and to send the related replies. A `ServicePanel` artifact is instantiated by specifying—as artifact configuration parameter—the URI of the WSDL containing the description of the service. The artifact usage interface provides two basic operations:

- `getNextRequestMsg`, used to retrieve incoming messages representing new operation requests to be served. Abstract signature:

  `getNextRequestMsg(filter:MsgFilter)`

  where `filter` can be specified to select the request messages to which the agent is interested to (for instance, related to a specific operation). The operation generates an observable event `new_op_request(msg:MsgInfo)` as soon as an message matching the filter is received by the artifact, containing information about the message arrived.

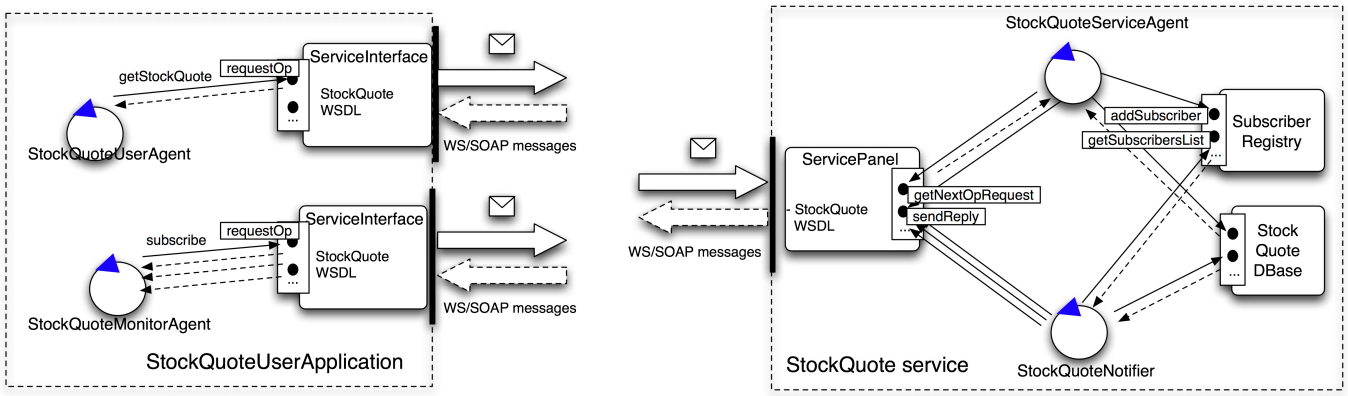

Fig. 6. An abstract representation of the stock quote user and service applications, with in evidence the agents and artifacts involved.

- `getMsgRelatedTo`, used to retrieve incoming messages related to previously sent messages. Abstract signature:

  ```
  getMsgRelatedTo(msgID:String)
  ```

  where `msgID` is the identifier of the message to which the reply is related. The operation generates an observable event `new_msg(msg:MsgInfo)` as soon as an message matching the filter is received by the artifact, containing information about the message arrived.
- `sendReply`, used to send message replies. Abstract signature:

  ```
  sendReply(toMsg:MsgInfo,
            msgContent:OMElement)
  ```

  where `toMsg` contains the information about the message to which the reply is related and `msgContent` is the content of the reply message (a XML piece of data, according to the XML schema described in the WSDL).

D. A simple example: Stock Quote with Agents and Artifacts

To give a concrete taste of simpA-WS, in the following we report the sketch of the implementation of a stock quote service, which is typically found among the basic examples of Web Service technologies: here we consider a slightly more articulated version, with a support not only for in-only and in-out (request-response) MEPs, but also out-only.

1) Stock Quote Service: The stock quote service is characterised by three basic operations (a sketch of the WSDL is reported in the appendix): `GetLastTradePrice`, an in-out (request-response) operation useful to get the current value of a stock given its name, `Subscribe`, an in-only (fire-and-forget) operation to subscribe the service in order to receive periodically the quote of a specified stock, and `NotifyTradePrice`, an out-only operation executed by the service to notify the value of a stock to a previously subscribed user. Figure 6 shows the architecture of a possible implementation of the service using simpA-WS: Table I, Table II, Table III and Table IV show the implementation of the service side, while Table V, Table VI and Table VII show the implementation of a simple application interacting with the service.

The service is composed by two types of agents: *(i)* `StockQuoteServiceAgent`, responsible for creating the service panel artifact and using it to process the incoming `GetLastTradePrice` and `Subscribe` requests; *(ii)* a `StockQuoteNotifierAgent`, responsible for periodically notifying its subscribers of the stock quotes. Besides these agents, two kinds of artifacts are exploited (other than `ServicePanel`): a `StockQuoteDBase` artifact, which functions as a store containing the stock quote values, and a `SubscribersRegistry`, which is used to keep track of the list of the service subscribers.

`StockQuoteServiceAgent` uses `ServicePanel` to process the requests as soon as they are collected: then, in the case of `GetLastTradePrice` requests, the agent accesses the database artifact to get the current value of the stock quote, and sends a reply with such information; instead, in the case of `Subscribe` requests, the agent registers the new subscribers by executing the `addSuscriber` operation on the `SubscribersRegistry` artifact. `StockQuoteNotifierAgent` periodically retrieves the subscribers' list by executing a `getSubscribers` operation on the `SubscribersRegistry` artifact: for each subscriber it then sends a `NotifyTradePrice` message with the updated value of the stock quote, retrieved from the database artifact. Table I and Table II report a sketch of the source code of the agents, Table III the skeleton of the artifacts, and Table IV the main of the service application (called `StockQuoteService`), which actually creates the initial set of agents and artifacts composing the application (apart the `ServicePanel` artifact, which is created by the `StockQuoteServiceAgent` agent).

Despite its simplicity, this example should give a quite concrete idea about how complex services—characterised, for instance, by complex message-exchange protocols, possibly including both pro-active and reactive parts, short and long-running activities, etc.—can be structured in terms of agents and artifacts.

2) Stock Quote User Application: Table V, Table VI and Table VII show an example of a simpA-WS user application exploiting Web services, in particular the stock quote service described by the WSDL reported in the Appendix. The application is com-

```
 public class StockQuoteServiceAgent
extends Agent {
  @ACTIVITY_WITH_AGENDA( {
    @TODO(activity="preparing"),
    @TODO(activity = "processingMsg",
                  pre="completed(preparing)", persistent = true),
  }) void main() {}

  @ACTIVITY void preparing() throws ActivityFailure {
    makeArtifact("StockQuoteService","alice.simpaws.ServiceInterface",
                           new ArtifactConfig("http://localhost:8080/axis2/wsdl/StockQuoteService.wsdl"));
  }

  @ACTIVITY void processingMsg() throws ActivityFailure {
    ArtifactId panel = lookupArtifact("StockQuoteService");
    SensorId sid = linkDefaultSensor();

    use(panel, new Op("getNextOpRequest"), sid);
    Perception p = sense(sid, "new_op_request", 300000);
    MsgInfo msg = (MsgInfo) p.getContent(0);
    String name = msg.getOperationName();

    if (name.equals("GetLastTradePrice")) {
      OMElement replyMsg = getStockValue(msg.getMsgContent());
      use(panel, new Op("sendReply", msg, replyMsg));
    } else if (name.equals("Subscribe")) {
      String stockName = getStockName(msg.getMsgContent());
      ArtifactId reg = lookupArtifact("subscribers-registry");
      use(reg,new Op("addSubscriber",msg,stockName));
    }
    unlinkSensor(sid);
  }

  private OMElement getStockValue(OMElement msg) {...}
  private String getStockName(OMElement elem){...}
}
```

TABLE I

STOCK QUOTE AGENT ON THE SERVICE SIDE

```
 public class StockQuoteNotifier
extends Agent {

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

  @ACTIVITY void notifyingStockQuotes() throws ActivityFailure {
    ArtifactId reg = lookupArtifact("subscribers-registry");
    SensorId sid = linkDefaultSensor();

    use(reg, new Op("getSubscribersList"), sid);
    Perception p = sense(sid, "subcribers_list");

    List<SubscriberInfo> list = (List<SubscriberInfo>) p.getContent();
    if (list.size()>0){
      ArtifactId panel = lookupArtifact("StockQuoteService");
      for (SubscriberInfo sinfo: list){
        OMElement replyMsg = getStockValue(sinfo.getStock());
        use(panel, new Op("sendReply", sinfo.getMsgSource(), replyMsg));
      }
    }
    unlinkSensor(sid);
    suspendActivityFor(2000);
   }

  private OMElement getStockValue(String stock) {...}
}
```

TABLE II

NOTIFIER AGENT ON THE SERVICE-SIDE

```
 public class SubscriberInfo {
  MsgContext msg;
  String quote;

  public SubscriberInfo(MsgContext msg, String quote){
    this.msg = msg;
    this.quote = quote;
  }
  public MsgContext getMsgSource(){ return msg; }
  public String getStock(){ return quote; }
}

public class SubscriberRegistry extends Artifact {
  private ArrayList<SubscriberInfo> list;

  public SubscriberRegistry(){
    list = new ArrayList<SubscriberInfo>();
  }

  @OPERATION void addSubscriber(MsgContext ctx, String quote){
    list.add(new SubscriberInfo(ctx,quote));
  }

  @OPERATION void getSubscribersList(){
    signal("subcribers_list", list.clone());
  }
}

public class StockQuoteDBase extends Artifact { ... }
```

TABLE III

ARTIFACTS USED ON THE SERVICE SIDE

```
 public class StockQuoteService extends
WSApplication {
  public void setup(){
    try {
      createArtifact("stock-quote-dbase",StockQuoteDBase.class);
      createArtifact("subscribers-registry",SubscriberRegistry.class);
      spawnAgent("stock-notifier-agent",StockQuoteNotifier.class);
      spawnAgent("stock-quote-agent",StockQuoteServiceAgent.class);
    } catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

TABLE IV

ENTRY POINT (MAIN) OF THE WEB SERVICE APPLICATION (SERVICE-SIDE)

```
 public class StockQuoteUserAgent
extends WSAgent {

  @ACTIVITY  void main() throws ActivityFailure {
    SensorId sid = linkDefaultSensor();
    ArtifactId ServiceInterface =
            makeServiceInterface("interface-1",
               "http://localhost:8080/axis2/wsdl/StockQuoteService.wsdl",
               "StockQuoteUserApplication");

    use(ServiceInterface,new Op("requestOp", "getStockQuote", makeGetStockQuoteMsg("ACME")),sid);
    Perception res = sense(sid, "msg_reply", 300000);
    OMElement replyMsg = (OMElement) res.getContent();

    log("Result: " + replyMsg.toString());
  }

  private  OMElement makeGetStockQuoteMsg(String stockName) {...}
  private void log(String st){...}
}
```

TABLE V

STOCK QUOTE USER AGENT ON THE CLIENT SIDE

```
 public class StockQuoteMonitorAgent
extends WSAgent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="subscribing"),
    @TODO(activity="collectingQuotes",
          pre="completed(subscribing)", persistent=true)
  }) void main(){}

  @ACTIVITY void subscribing() throws ActivityFailure {
    SensorId sid = linkDefaultSensor();
    ArtifactId ServiceInterface =
            makeServiceInterface("interface-2",
               "http://localhost:8080/axis2/wsdl/StockQuoteService.wsdll",
               "StockQuoteUserApplication");

    OMElement subscribeMsg = makeRegisterMsg("ACME"));
    use(ServiceInterface,new Op("sendMsg","Subscribe",subscribeMsg),sid);
    Perception p = sense(sid, "msg_sent|msg_send_failed", 5000);
    if (p.getLabel().equals("msg_sent")){
      memo("subscribe_message_id",(String)p.getContent());
      unlinkSensor(sid);
    } else {
      throw new ActivityFailure();
    }
  }

  @ACTIVITY void collectingQuotes() throws ActivityFailure {
    SensorId sid = linkDefaultSensor();
    ArtifactId ServiceInterface = lookupArtifact("interface-2");
    String msgId = (String)getMemo("subscribe_message_id").getArg(0);

    use(ServiceInterface, new Op("getReply", msgId), sid);
    Perception res = sense(sid, "msg_reply", 300000);
    OMElement replyMsg = (OMElement) res.getContent();

    log("New quote value: "+replyMsg.toString());
    unlinkSensor(sid);
  }

  private  OMElement makeRegisterMsg(String stockName) {...}
  protected void log(String st){...}
}
```

TABLE VI

STOCK QUOTE MONITOR AGENT ON THE CLIENT-SIDE

```
 public class StockQuoteUserApplication
extends WSApplication {
  public void setup(){
    try {
      spawnAgent("stock-quote-user-agent",StockQuoteUserAgent.class);
      spawnAgent("stock-quote-monitor-agent",StockQuoteMonitorAgent.class);
    } catch (Exception ex){
      ex.printStackTrace();
    }
  }
}
```

TABLE VII

ENTRY POINT (MAIN) OF THE WEB SERVICE APPLICATION (CLIENT-SIDE)

posed by two agents, `StockQuoteUserAgent` and `StockQuoteMonitorAgent`. The former is characterised by a simple main activity, which creates a proxy artifact for interacting with the stock quote service—in particular, to request the `GetLastTradePrice` operation (specifying `ACME` as stock quote name), and get the reply, which is then logged to the standard output. The latter has a slightly more complex behaviour: first, it creates a service interface artifact to interact with the stock quote service (alternatively, a single service interface artifact could be shared and used by the two agents), and subscribes (via the `Subscribe` operation) to receive a periodic notification. The agent then starts its cyclic `collectingQuotes` activity, receiving and logging all the notifications sent back by the service. A sketch of the source code of the agents is reported in Table V and Table VI, while Table VII shows the main of the application (called `StockQuoteUserApplication`), which simply spawns an instance of both agents.

VI. CONCLUSIONS

In this paper we introduced an agent-oriented programming model for developing SOA/WS applications, based on the A&A meta-model, and simpA-WS, a framework built on top of the Java platform for concretely programming service and user applications in terms of agents and artifacts. This approach contrasts with the choices adopted by leading software

vendors in the state-of-the-art, which is typically based on a component-oriented programming model. This paper extends a previous work [14], with the description of the simpA-WS framework.

Actually, as widely reported in literature [10], Service-Oriented Computing and Web Services are considered among the most promising and important application contexts for agents and MAS. However, the focus of existing agent and MAS research approaches is typically on exploiting AI-related features and techniques for supporting the dynamic discovery and flexible composition and orchestration of services. In this paper, instead, we focussed on programming and software engineering issues, highlighting the value of agent-oriented abstractions as basic building blocks for designing and programming services and applications using services.

Indeed the programming model introduced and the related technologies—such as simpA-WS—are still in their infancy and further work is needed along several directions. In the paper we have just considered the very basic issues concerning SOA and Web Services, without dealing with other important advanced topics such as quality of service (security, reliability, trust,...), service composition and coordination (orchestration, choreography, transactions), along with the related WS-* specifications (WS-Security, WS-Trust, WS-Coordination, etc). Future work will then be devoted to frame such issues in the SA&A programming model, and to stress the approach with real-world case studies and applications, besides the simple toy examples included in the current distribution. In particular, as a concrete case-study we will consider the sample application provided in the WS-I web site[8], trying to compare our approach with those of the leading software vendors (IBM, Microsoft, Sun, SAP—to mention some) available on such site.

REFERENCES

[1] The aliCE Research Group. simpA official web site. http://www.alice.unibo.it/projects/simpa.

[2] Sriram Anand, Srinivas Padmanabhuni, and Jai Ganesh. Perspectives on service oriented architectures. In *Proceedings of the 2005 IEEE International Conference on Service Computing*, volume 2. IEEE, 2005.

[3] Nick Benton, Luca Cardelli, and Cedric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[4] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[5] IBM et al. Service component architecture. http://www-128.ibm.com/developerworks/library/specification/ws-sca/, 2006.

[6] Rafael Bordini et. al. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

[7] Donald F. Ferguson and Marcia L. Stockon. Service-oriented architecture: Programming model and product architecture. *IBM Systems Journal*, 44(4):753–780, 2005.

[8] Sun Microsystems. The java API for XML web services (JAX-WS 2.0). http://java.sun.com/webservices/jaxws/.

[9] Sun Microsystems. Service oriented business integration. http://java.sun.com/integration/.

[10] Michael N. Huhns, Munindar P. Singh, Mark Burstein, Keith Decker, Edmund Durfee, Tim Finin, Les Gasser, Hrishikesh Goradia, Nick Jennings, Kiran Lakkaraju, Hideyuki Nakashima, H.Van Dyke Parunak, Jeffrey S. Rosenschein, Alicia Ruvinsky, Gita Sukthankar, Samarth Swarup, Katia Sycara, Milind Tambe, Tom Wagner, and Laura Zavala. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, November 2005.

[11] Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.

[12] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Sciences*, 150(3):21–36, 29 May 2006.

[13] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA, 19–23 July 2004. ACM.

[14] Alessandro Ricci, Claudio Buda, and Nicola Zaghini. An agent-oriented programming model for soa & web services. In *5th IEEE International Conference on Industrial Informatics (INDIN'07). Special Session on Agents Theory and Practice for Industry (ATPI)*, Vienna, Austria, July 2007.

[15] Alessandro Ricci and Mirko Viroli. simpA: An agent-oriented approach for prototyping concurrent applications on top of java. In *Proceedings of the ACM International Conference on Principle and Practice of Programming in Java (PPPJ'07)*, Lisboa, Portugal, September 2007. ACM.

[16] Clemens Szyperski. *Component Software*. Addison-Wesley Professional, November 2002.

[17] W3C WS Working Group. Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[8] http://www.ws-i.org/

APPENDIX

This appendix reports a sketch of the WSDL code of the stock quote service discussed in the text.

```
 <?xml version="1.0"?>
 <definitions name="StockQuote"
              targetNamespace="http://example.com/stockquote.wsdl"
              xmlns:tns="http://example.com/stockquote.wsdl"
              xmlns:xsd1="http://example.com/stockquote.xsd"
              xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
              xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
         <complexType>
           <all>
             <element name="price" type="float"/>
           </all>
         </complexType>
      </element>
      ...
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>
  <message name="SubscribeMsg"> ... </message>
  <message name="NotifyTradePriceMsg"> ... </message>

  <portType name="StockQuotePortType">

    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>

    <operation name="Subscribe">
      <input name="Subscribe" message="tns:SubscribeMsg"/>
    </operation>

    <operation name="NotifyTradePrice">
      <output name="NotifyTradePrice" message="tns:NotifyTradePriceMsg"/>
    </operation>

  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    ...
  </binding>

  <service name="StockQuoteService">
    <documentation>Stock quote service example </documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>

</definitions>
```