FEATHERWEIGHT AGENT LANGUAGE
A Core Calculus for Agents and Artifacts

Ferruccio Damiani
Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italy

Paola Giannini
Dipartimento di Informatica, Università del Piemonte Orientale, Via Bellini 25/G, 15100 Alessandria, Italy

Alessandro Ricci, Mirko Viroli
DEIS, Università di Bologna, Via Venezia 52, 47023 Cesena, Italy

Keywords: Multi-agent systems, Concurrency, Core calculi, Type systems.

Abstract: The widespread diffusion and availability of multicore architectures is going to make more and more aspects of concurrency and distribution to be part of mainstream programming and software engineering. The SIMPA framework is a recently proposed library-based extension of JAVA that introduces on top of the OO layer a new abstraction layer based on agent-oriented concepts. A SIMPA program is organized in terms of dynamic set of autonomous pro-active task-oriented entities – the *agents* – that cooperate by exploiting some *artifacts*, that represents resources and tools that are dynamically constructed, shared and co-used by agents. In this paper we promote the applicability of the agent and artifact metamodel in OO programming a step further. Namely, we propose a core calculus that integrates techniques coming from concurrency theory and from OO programming languages to provide a first basic formal framework for designing agent-oriented languages and studying properties of agent-oriented programs.

1 INTRODUCTION

Multi-core architectures, Internet-based computing and Service-Oriented Architectures/Web Services, are increasingly introducing concurrency issues (and distribution) in the context of a large class of applications and systems—up to making them key factors of almost *any* complex software system. As noted in (Sutter and Larus, 2005), even though concurrency has been studied for about 30 years in the context of computer science fields such as programming languages and software engineering, this research has not significantly impacted on mainstream software development. However, it appears more and more important to introduce higher-level abstractions, which can “help build concurrent programs, just as object-oriented abstractions help build large component-based programs” (Sutter and Larus, 2005).

The A&A (Agents and Artifacts) meta-model, recently introduced in the context of agent-oriented programming and software engineering as a novel foundational approach for modelling and engineering complex software systems (Omicini et al., 2009), goes in this direction. *Agents* and *artifacts* are the basic high-level and coarse-grained abstractions available in A&A: agents are used to model (pro)-active and task-oriented components of a system, which encapsulate the logic and control of their execution, while artifacts model purely-reactive function-oriented components of a system, used by agents to support their (invidual and collective) activities.

In (Ricci and Viroli, 2007) it is introduced SIMPA, a library-based extension of JAVA providing programmers with *agent-oriented abstractions* on top of the basic OO layer, to be used as basic building blocks to define the architecture of complex (concurrent) applications. In SIMPA, the underlying OO computational model of JAVA is still adopted, but only for defining agents and artifacts programming and data storage, namely, for defining the purely computational part of

Damiani F., Giannini P., Ricci A. and Viroli M. (2009).
FEATHERWEIGHT AGENT LANGUAGE - A Core Calculus for Agents and Artifacts.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 218-225
DOI: 10.5220/0002257102180225

applications. On the other hand, agents and artifacts are used to define aspects related to system architecture, interaction, and synchronisation.

In this paper we promote the applicability of A&A metamodel in OO programming a step further, by introducing FAL (FEATHERWEIGHT AGENT LANGUAGE), a core calculus formalizing the key features of SIMPA. The formalization is largely inspired to FJ, (FEATHERWEIGHT JAVA) (Igarashi et al., 2001), and is based on reduction rules applied at certain evaluation contexts. On the other hand, being concurrency-oriented, this calculus uses techniques coming from concurrency theory, as e.g. in process algebras. A system configuration is seen as a parallel composition of agents and artifacts instances (seen as independent and asynchronous processes), the former keeping track of a tree of (sub-)activities to be executed in autonomy, the latter holding a set of pending operations to be executed in response to agent actions over the artifact.

Organisation of the Paper. Section 2 introduces the SIMPA programming model. Section 3 presents syntax, and operational semantics of the FAL calculus. Section 4 briefly discusses the properties that result from type soundness. Section 5 discusses some related work and Section 6 concludes by outlining possible directions for further work.

2 THE PROGRAMMING MODEL

In this section we describe an abstract version of SIMPA programming model by exploiting the syntax of the FAL calculus.

The Agent Programming Model. In essence, an agent in SIMPA is a stateful entity whose job is to pro-actively execute a structured set of *activities* as specified by the agent programmer, including possibly non-terminating activities, which finally result in executing sequences of *actions*, either internal actions – inspecting/changing its own state – or external actions – interacting with its environment. All actions are executed atomically.

The state of an agent is represented by an associative store, called *memo-space*, which represents the long-term memory 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). For instance, the philosopher agent uses a memo `hungry` to take note that its state is now *hungry* and it needs the forks, and `stopped` to keep track that it needs to terminate. A basic set of internal actions is available to agents to work atomically with the memo-space: `+memo` is used to create a new memo with a specific label and a variable number of arguments, `?memo` and `-memo` to get/remove a memo with the specified label.

```
agent Main {
  activity main() { Table t = make Table(new boolean[5]);
    spawn Philosopher(0,1,t); spawn Philosopher(1,2,t);
    spawn Philosopher(2,3,t); spawn Philosopher(3,4,t);
    spawn Philosopher(4,0,t); }
}
artifact Table { boolean[] isBusyFork;
  operation getForks(int left, int right)
   :guard ((not(.isBusyFork[left]) and (not(.isBusyFork[right])))
  { .isBusyFork[left] = true; .isBusyFork[right] = true;
    signal(forks_acquired);
  }
  operation releaseForks(int left, int right) :guard true
    { .isBusyFork[left] = false; .isBusyFork[right] = false; }
}
agent Philosopher { Sns s;
  activity main(int left, int right, Table table)
      :agenda (  prepare() :pre true,
                 living(left,right,table) :pre memo(hungry)
                      :pers not(memo(stopped)) ) { }
  activity prepare() { +memo hungry; }
  activity living(int left, int right, Table table)
      :agenda (  eating(left,right,table) :pre memo(hungry),
                 thinking() :pre completed(eating),
                 shutdown() :pre failed(eating)  ) { }
  activity thinking() { ... /* think */  +memo hungry; }
  activity eating(int left, int right, Table table)
  { use table.getForks(left,right) :sns s
    sense s :filter forks_acquired;
    ... /* eat */
    use table.releaseForks(left,right);
    -memo(hungry);
  }
  activity shutdown() { +memo(stopped); }
}
```

Figure 1: The five dining philosophers problem.

The computational behaviour of an agent can be defined as a hierarchy of activities (corresponding to the execution of some tasks). Activities can be simple or structured. A simple activity is composed by just a flat sequence of actions, as a single control flow, while structured activities have a non-empty agenda specifying sub-activities, which in turn can be possibly executed in the context of such super-activity—hence leading to the hierarchical structure of behaviour. At the language level, simple activities are represented by `activity` blocks, providing the name of the activity and parameters. By default each agent has a `main` activity, which can be either simple or structured. In the dining philosophers example shown in Figure 1, the `Philosopher` agent has the simple activities, `prepare`, `eating`, `thinking`, and `shutdown`. A structured activity has a non-empty *agenda*, specifying a set of *todos* representing sub-activities that must be executed in the context of the parent activity—also called super-activity. In the philosophers example, `main` and `living` are structured activities. A todo contains the name of the sub-activity to be executed, a precondition over the inner state of the agent that must be hold for the specified sub-activity to start, and attributes related to sub-activity execution, such as persistency. Preconditions are expressed as a boolean expression 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 (if they started, completed, or aborted) and on *(ii)* the local inner state of the agent, that is the memo space. For instance, the predicate `memo(M)` is true if the specified memo `M` is found in the memo space. In the example, in the structured activity `living`, sub-activity `eating` is executed as soon as a memo `hungry` is found in the memo space. When the precondition of a todo item holds (for an activity in execution listing such todo in the agenda), the todo is removed from the agenda and an instance of the sub-activity is created and executed. So, multiple sub-activities can be executed concurrently and asynchronously, in the context of the same parent activity. Sub-activities execution can be then synchronized by properly specifying preconditions in todos, hence in a declarative way. If a todo is declared persistent, as soon as the sub-activity is completed the todo is re-inserted into the agenda. The persistency attribute can specify also the condition under which the activity should persist. For instance, the todo item about `living` sub-activity in philosopher agent is declared persistent until a `stopped` memo is found.

The Artifact Programming Model. An artifact is composed by three main parts: *(i)* observable properties, which are attributes that can be observed by agents without an explicit agent action towards the artifact; *(ii)* a description of the inner non-observable state, composed by set of state variables analogous to private instance fields of objects; and *(iii) operations*, which embody the computational behaviour of artifacts. The `Table` artifact in the philosopher example in Figure 1 has no observable properties, an inner state variable `isBusyFork`, an array of booleans, and two operations, `getForks` and `releaseForks`, the first used to acquire two forks and the latter for releasing forks. Both state variables and observable properties are declared similarly to instance fields in objects; observable properties are prefixed by `obsprop` qualifier. In both cases, a dot notation (e.g. `.isBusyFork`) is used both for l-value and r-value, to syntacatically distinguish them from parameters.

Operations can be defined by method-like blocks qualified as `operation`, specifying the name and parameters of the operation and a computational body. It is worth noting that no return parameter is specified, since operations in artifacts are not exactly like methods in objects. For each `operation`, implicitly an *interface control* in the usage interface is defined, with the specified signature. Operations can be either *atomic*, executed as a single computational step, or *structured*, i.e. composed by multiple atomic operation steps. For sake of space, in this paper we consider only atomic operations. For each operation a *guard* can be specified (`:guard` declaration), representing the condition that must hold for the related control in the usage interface to be enabled. For instance, the `getForks` operation in `Table` artifact is available – i.e. the related control is enabled in the usage interface – when the specified forks are not busy.

To be useful, an artifact typically should provide some level of *observability*. This is achieved both by generating observable events through the `signal` primitive, and 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 labelled tuple, whose label represents the kind of the event and the information content. For instance, in the `Table` artifact `getForks` operation generates the `forks_acquired(Left,Right)` tuple. Actually, the observable event `op_exec_completed` is automatically generated – without explicit `signals` – as soon as the execution of an operation is completed. In the latter case, observable properties are instance variables qualified as `obsprop`. Any time the property changes, an observable event of type `prop_updated` is fired 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 details in next subsection). An example of simple artifact with observable properties is the `Counter` artifact shown in Figure 2: this artifact – working as an observable counter – has just a single observable property named `count` and an `inc` operation to update this count. Each time the operation is executed, the observable property and the event `prop_update(count,Value)` are automatically generated.

The Agent-artifact Interaction Model. As already stated, artifact *use* and *observation* are the basic form of interaction between agents and artifacts. Artifact use by an agent involves two basic aspects: *(i)* executing operations on the artifact, and *(ii)* perceiving through agent *sensors* the observable events generated

```
artifact Counter { obsprop int count;
  Counter(int c){ .count = c; }
  operation inc() { .count = .count+1; }
}
agent Main {
  activity main() {
    Counter c = make Counter(0);
    spawn Observer(c); spawn User(c); spawn User(c); }
}
agent Observer { Sns s;
  activity main(Counter c)
           :agenda (  prepare(c),
                      monitoring(c) :pre completed(prepare)
                          :pers (not memo(finished)) { }
  activity prepare(Counter c) { focus (c,s); }
  activity monitoring(Counter c) {
    sense s :filter prop_updated;
    int value = observe c.count;
    ... // do something
    if (value >= 100 ){ +memo(finished); } }
}
agent User {
  activity main(Counter c)
          :agenda (  usingCount(c) :pers true  ) {}
  activity usingCount(Counter c) { use c.inc(); }
}
```

Figure 2: A simple program with an `Observer` agent continuously observing a `Counter` Artifact, which is concurrently used by two `User` agents.

by the artifact. Conceptually sensors represent a kind of "perceptual memory" of the agent, used to detect events coming from the environment, organize them according to some policy – e.g. FIFO and priority-based – and finally make them available to the agent. In the abstract language presented here, sensors used by an agent are declared at the beginning of the `agent` block.

In order to trigger operation execution, the `use` action is provided, specifying the target artifact, the operation to execute – or, more precisely, the usage interface control to act upon, which activates the operation – and optionally, a timeout and the identifier of the sensor used to collect observable events generated by the artifact. The action is blocked until either the action execution succeeds – which means that the specified interface control has been finally selected and the related operation has been started – or fails, either because the specified usage interface control is invalid (for instance it is not part of the usage interface) or the timeout occurred. If the action execution fails an exception is generated. In the philosopher example, a `Philosopher` agent (within its `eating` activity) executes a `use` action so as to execute the `getForks` operation, specifying the `s` sensor. On the artifact side, if the forks are busy the `getForks` usage interface control is not enabled, and the `use` is suspended. As soon as the forks become available the operation is executed and the `use` action succeeds.

It is important to note that no control coupling exists between an agent and an artifact while an operation is executed. However, operation triggering is a synchronization point between the agent (user) and the artifact (used): if the use action is successfully executed, then this means that the execution of the operation on the artifact has started.

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 is returned. In the philosopher example, after executing `getForks` the philosopher agent blocks until a `forks_acquired` event is perceived on the sensor `s`. If no perception are sensed for the duration of time specified, the action generates an exception. Pattern-matching can be tuned by specifying custom event-selection filter: the default filter is based on regular-expression patterns, matched over the event type (a string).

Besides sensing events generated when explicitly using an artifact, 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 actions 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 filter to define the set of events to observe. The latter one is used to stop observing the artifact. In the example shown in Figure 2, an `Observer` agent continuously observes a `Counter` artifact, which is used by two `User` agents. After executing a `focus` on the artifact in the `prepare` activity, in the `monitoring` activity the observer prints on a console artifact the value of the observable property `count` as soon as it changes.

3 THE CORE CALCULUS

The syntax of FAL is summarised in Figure 3 where we assume a set of basic values, ranged over by the metavariable `c`. Types for basic values are ranged over by the metavariable `C`. We only assume the basic values `true` and `false` (of type `Bool`) which are used as the result of the evaluation of preconditions, persistency predicates and guards. We use the overbar se-

U	::=	G $\mid$ A $\mid$ C	Agent / artifact / basic value types
T	::=	U $\mid$ Sns	Types
GD	::=	agent G { Sns $\overline{s}$; $\overline{Act}$ }	Agent (class) definition
Act	::=	activity a ($\overline{T\ x}$) :agenda ($\overline{SubAct}$) {e; }	Activity definition
SubAct	::=	a($\overline{e}$) :pers e :pre e	Subactivity definition
AD	::=	artifact A {$\overline{U\ f}$; $\overline{U\ p}$; $\overline{Op}$ }	Artifact (class) definition
Op	::=	operation o ($\overline{U\ x}$) :guard e {e; }	Operation definition
e	::=	x $\mid$ c	Expressions: variable / basic value
	$\mid$	spawn G($\overline{e}$) $\mid$ make A($\overline{e}$)	agent and artifact instance creation
	$\mid$	e; e	sequential composition
	$\mid$	.f $\mid$ .f = e	artifact-field access / update
	$\mid$	.p $\mid$ .p = e	artifact-property access / update
	$\mid$	signal(l(e))	event generation
	$\mid$	.s $\mid$ use e.o($\overline{e}$) :sns e $\mid$ sense e :filter l	sensor / operation use / event sensing
	$\mid$	focus(e, e) $\mid$ unfocus(e, e)	focus / unfocus
	$\mid$	observe e.p	get property value
	$\mid$	?memo(l) $\mid$ -memo(l) $\mid$ +memo(l(e))	memo operations
	$\mid$	memo(l)	memo predicate
	$\mid$	started(a) $\mid$ completed(a) $\mid$ failed(a)	activity state predicates
	$\mid$	fail	activity error

Figure 3: Syntax.

quence notation according to (Igarashi et al., 2001).

There are minor differences between the syntax of the calculus and the one of the language used for the examples. Namely: instead of tuples for memos in memo-spaces (and event in sensors) we use values; and specifiers (:agenda, :pers, :pre, :guard and :sns), that are optional in the language, are mandatory in the calculus.

Labels are used as keys for the associative maps representing the content of sensors and memo-spaces. The metavariable l range over labels.

The expression fail model failures in activities, such as the evaluation of ?memo(l) and -memo(l) in an agent in which the memo-space does not have a memo with label l. Note that the types of parameters, in artifact operations and the type of fields and properties may not be sensors so artifacts. Moreover, the signal expression, signal(l(e)), does not specify a sensor. Therefore, sensors may not be explicitly manipulated by artifacts.

The language is provided with a standard type system enforcing the fact that expressions occur in the right context (artifact or agent), operation used, and activities mentioned in todo lists are defined, and only defined fields and properties are accessed/modified.

Operational Semantics. The operational semantics is described by means of a set of reduction rules that transform sets of instances of agents/artifacts/sensors.

Each agent/artifact/sensor instance has a unique identity, provided by a *reference*. The metavariable γ ranges over references to instance of agents, α over artifacts, σ over sensors. *Configurations* are non-empty sets of agent/artifact/sensor instances.

Sensor instances are represented by $\sigma = \langle \overline{l\ v} \rangle^{Sns}$, where σ is the instance identifier, and $\overline{l\ v}$ is the queue of association labels/values representing the events generated (and not yet perceived) on the sensor.

Agent instances are represented by $\gamma = \langle \overline{l\ v}, \overline{\sigma}, R \rangle^{G}$, where γ is the agent identifier, G is the type of the agent, $\overline{l\ v}$ is the content of the *memo-space*, $\overline{\sigma}$ is the sequence of references to the instances of the sensors that the agent uses to perceive, and R is the state of the activity, main, that was started when the agent was created. The sensor instances in $\overline{\sigma}$ are in one-to-one correspondence with the sensor variables declared in the agent and are needed since every agent uses it own set of sensor instances.
An *instance of an activity*, R, describes a running activity. As explained in Section 2, before evaluating the body of an activity we have to complete the execution of its sub-activities, so we also represent the state of execution of the sub-activities.

$$R ::= a(\overline{v})[Sr_1 \cdots Sr_n]\{e\} \mid \mathtt{failed}^{a}$$

The name of the activity is a, $\overline{v}$ are the actual parameters of the current activity instance, $Sr_1 \cdots Sr_n$ is the set of sub-activities running, and e is the state of evaluation of the body of the activity. (Note that the evaluation of the body starts only when all the sub-activities have been fully evaluated.) With $\mathtt{failed}^{a}$ we say that activity a has *failed*. If the evaluation of a sub-activity is successful then it is removed from the

set $\mathtt{Sr}_1 \cdots \mathtt{Sr}_n$. So when $n = 0$ starts the evaluation of the body e.
For a sub-activity, Sr, the process of evaluating its precondition (we do not consider the persistency predicate that would be similar), is represented by the term, $\mathtt{a}(\overline{\mathtt{v}})\langle\mathtt{e}\rangle$ where e is different from true or false (it is the state of evaluation of the precondition) when e = true, the term $\mathtt{a}(\overline{\mathtt{v}})\langle\mathtt{true}\rangle$ is replaced with the initial state of the evaluation of the activity a with parameters $\overline{\mathtt{v}}$. When e = false the evaluation of the precondition of a is rescheduled. Therefore:

$$\mathtt{Sr} \ ::= \ \mathtt{a}(\overline{\mathtt{v}})\langle\mathtt{e}\rangle \ \mid \ \mathtt{R}$$

Artifact instances are represented by $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_n\rangle^{\mathtt{A}}$ where α is the artifact identifier, A the type of the artifact, the sequence of pairs $\overline{\mathtt{f}}\,\overline{\mathtt{v}}$ associates a value to each the field of A, the sequence of pairs $\overline{\mathtt{p}}\,\overline{\mathtt{w}}$ associates a value to each property of A, the sequence $\overline{\sigma}$ represents the sensors that agents focusing on A are using, and $\mathtt{O}_i$, $1 \le i \le n$, are the operations that are in execution. We consider $\mathtt{O}_1 \cdots \mathtt{O}_n$ a queue with first element $\mathtt{O}_n$ and last $\mathtt{O}_1$. (For simplicity, we do not consider steps in this paper, although we have a full formalization including them.) Artifacts are single threaded and (differently from agents that may have more activity running at the same time) only the operation $\mathtt{O}_n$ is being evaluated.
A *running operation*, O, is defined as follows.

$$\mathtt{O} \ ::= \ (\sigma, \mathtt{o}\langle\mathtt{e}\rangle\{\mathtt{e}'\})$$

where σ identifies the sensor associated with the operation which was specified by the agent containing the use that started the operation, and that is used to collect events generated during the execution of the operation by signal. If the expression $\langle\mathtt{e}\rangle$ is different from true or false the operation is evaluating its guard e. If e = true then the operation is evaluating its body. If e = false then the operation is removed from the queue and put at the end of it so that when it will be rescheduled it will restart evaluating its guard.

Reduction Rules by Examples. The initial configuration for the program in Fig. 1 is:[1]

$$\gamma_{\mathtt{Main}} = \langle\emptyset,\emptyset,\mathtt{main}\left[\begin{array}{l}\boxed{\mathtt{Table\,t=make\ Table(newBool[5]);}}\\ \mathtt{spawn\ Philosopher(0,1,t);}\\ \cdots\\ \mathtt{spawn\ Philosopher(4,0,t)}\end{array}\right]\{\,\}\rangle^{\mathtt{Main}}$$

The expression new Bool[5] reduces to the array [f,f,f,f,f] (In the array we use f for false and t for true.) Then the expression make Table([f,f,f,f,f]) reduces to an artifact reference α and adds to the configuration the initial artifact instance that follows:

$$\alpha = \langle .\mathtt{isBF} = [\mathtt{f,f,f,f,f}],\emptyset,\emptyset,\emptyset\rangle^{\mathtt{Table}}$$

After the initialization of the local variable t the agent instance $\gamma_{\mathtt{Main}}$ becomes

$$\gamma_{\mathtt{Main}} = \langle\emptyset,\emptyset,\mathtt{main}\left[\begin{array}{l}\boxed{\mathtt{spawn\ Philosopher(0,1,\alpha);}}\\ \cdots\\ \mathtt{spawn\ Philosopher(4,0,\alpha)}\end{array}\right]\{\,\}\rangle^{\mathtt{Main}}$$

The five spawn expressions are evaluated from left to right. The evaluation of the expressions spawn Philosopher(0,1,α) reduces to γ_0 and adds to the configuration the agent instance

$$(1)\quad \gamma_0 = \langle\emptyset,\sigma_0,\mathtt{main}\left[\begin{array}{l}\boxed{\mathtt{prepare()}\ \langle\mathtt{true}\rangle},\\ \mathtt{living(0,1,\alpha)}\ \langle\mathtt{memo(hungry)}\rangle\end{array}\right]\{\,\}\rangle^{\mathtt{Phil.}}$$

and the sensor instance $\sigma_0 = \langle\emptyset\rangle^{\mathtt{Sns}}$.
Similarly, the reduction of the other spawn expressions generates four agent instances and four sensor instances producing the configuration:

$$\gamma_{\mathtt{Main}} = \langle\ldots\rangle \quad \sigma_0 = \langle\emptyset\rangle \cdots \sigma_4 = \langle\emptyset\rangle \quad \alpha = \langle\ldots\rangle \quad \gamma_0 = \langle\ldots\rangle \cdots \gamma_4 = \langle\ldots\rangle$$

in which the agent $\gamma_{\mathtt{Main}}$ is inactive, having finished the evaluation of its body. The artifact α does not have any pending operation, and all the agent philosophers may start the execution of the sub-activities of their main activity (by starting the evaluation of the preconditions of prepare and living). Our modeling make use of nondeterministic evaluation rules, but parallel execution could be modeled.
Going back to (1), since the precondition of the runtime sub-activity prepare() of the activity main of the agent γ_0 is true the expression $\mathtt{prepare()}\ \langle\mathtt{true}\rangle$ is replaced by $\mathtt{prepare()}[\,]\{+\mathtt{memo(hungry)}\}$ (whose evaluation causes the insertion of the label hungry into the memo of γ_0) and then since the body is fully evaluated prepare is removed from the sub-activities of main , yielding

$$(2)\quad \gamma_0 = \langle\mathtt{hungry},\sigma_0,\mathtt{main}\left[\ \mathtt{living(0,1,\alpha)}\ \boxed{\langle\mathtt{memo(hungry)}\rangle}\ \right]\{\,\}\rangle^{\mathtt{Phil.}}$$

If instead of evaluating the sub-activity prepare we would have evaluated the precondition of the sub-activity living, the result would have being

$$\gamma_0 = \langle\emptyset,\sigma_0,\mathtt{main}\left[\begin{array}{l}\mathtt{prepare()}\ \langle\mathtt{true}\rangle,\\ \mathtt{living(0,1,\alpha)}\ \langle\mathtt{false}\rangle\end{array}\right]\{\,\}\rangle^{\mathtt{Phil.}}$$

Next time the sub-activity living was scheduled for execution $\mathtt{living(0,1,\alpha)}\ \langle\mathtt{false}\rangle$ would have been replaced with $\mathtt{living(0,1,\alpha)}\ \langle\mathtt{memo(hungry)}\rangle$.
Continuing from (2) the precondition memo(hungry) of living evaluates to true and the sub-activity $\mathtt{living(0,1,\alpha)}\ \langle\mathtt{true}\rangle$ is replaced by the corresponding run-time activity resulting in the following:

$$\mathtt{living(0,1,\alpha)}\left[\begin{array}{l}\mathtt{eating(0,1,\alpha)}\boxed{\langle\mathtt{memo(hungry)}\rangle},\\ \mathtt{thinking()}\ \langle\mathtt{completed(eating)}\rangle,\\ \mathtt{shutdown()}\ \langle\mathtt{failed(eating)}\rangle\end{array}\right]\{\,\}$$

The precondition memo(hungry) evaluates to true and the sub-activity $\mathtt{eating(0,1,\alpha)}\langle\mathtt{true}\rangle$ is replaced by the corresponding run-time activity resulting in

[1]The syntax of FAL does not include local variables and array object values. In this example, we will handle the local variable t by replacing, after its declaration/inizialization, all its occurrences with its value.

the following

$$(3)\quad \texttt{eating(0,1,}\alpha\texttt{)[ ]}\left\{\begin{array}{l}\boxed{\texttt{use }\alpha\texttt{.getForks(0,1) :sns }\sigma_0};\\ \texttt{sense }\sigma_0\texttt{ :filter forks\_acquired};\\ \texttt{/* eat */}\\ \texttt{use }\alpha\texttt{.releaseForks(0,1)};\\ \texttt{-memo(hungry)}\end{array}\right\}$$

(Note that both `completed(eating)` and `failed(eating)` would evaluate to `false`.) The evaluation of the body of `eating` can now start by reducing the expression use α.getForks(0,1) :sns σ_0, that schedules the operation `getForks` in the artifact instance α yielding

$$\alpha = \langle \texttt{.isBF} = \texttt{[f,f,f,f,f]},\emptyset,\emptyset,(\sigma_0,\texttt{getForks}\ \langle e'_0\rangle\ \{e_0\})\rangle^{\texttt{Table}}$$

where e'_0 is `(not(.isBF[0]) and (not(.isBF[1])))` and e_0 is

`.isBF[0] = true; .isBF[1] =true; signal(forks_acquired).`

The guard e'_0 reduces to `true`. The reduction of e_0 updates the array ι to `[t, t, f, f, f]` and adds the label `forks_acquired` to the queue of events of the sensor instance σ_0, yielding $\sigma_0 = \langle\texttt{forks\_acquired}\rangle^{\texttt{Sns}}$. Other agents may schedule operation the artifact α. For instance, if the agent γ_1 and γ_2 invoke the operation `getForks` on α, when the evaluation of `getForks` for the agent γ_0 was completed the state of the artifact would be

$$\alpha = \langle \texttt{.isBF} = \texttt{[t,t,f,f,f]},\emptyset,\emptyset,\\ (\sigma_2,\texttt{getForks}\ \langle e'_2\rangle\ \{e_2\})\ (\sigma_1,\texttt{getForks}\ \langle e'_1\rangle\ \{e_1\})\rangle^{\texttt{Table}}$$

So the guard e'_1 (`( not(.isBF[1]) and (not(.isBF[2])) )`) would evaluate to `false`, and the associated operation would be rescheduled and put at the rear of the queue yielding the following

$$\alpha = \langle \texttt{.isBF} = \texttt{[t,t,f,f,f]},\emptyset,\emptyset,\\ (\sigma_1,\texttt{getForks}\ \langle e'_1\rangle\ \{e_1\})\ (\sigma_2,\texttt{getForks}\ \langle e'_2\rangle\ \{e_2\})\rangle^{\texttt{Table}}$$

so the evaluation of the guard of the `getForks` operation invoked by γ_2 may start (and will successfully acquire the forks for γ_2). At the same time, the expression sense σ_0 :filter forks_acquired in (3) could be evaluated, perceiving the event `forks_acquired` and removing it from the sensor instance σ_0 which becomes $\sigma_0 = \langle\emptyset\rangle^{\texttt{Sns}}$. The code "`/* eat */`" may be executed and, at the end of its execution the expression use α.releaseForks(0,1) schedules the operation `releaseForks` on the artifact α and then `-memo(hungry)`removes the label `hungry` from the memo completing the execution of the sub-activity `eating`. The sub-activity `eating` is discarded and therefore the predicate `completed(eating)` becomes true and the sub-activity `thinking` could be executed resulting in γ_0 to be:

$$\langle\emptyset,\sigma_0,\texttt{main[living(0,1,}\alpha\texttt{)}\left[\begin{array}{l}\texttt{thinking()[]\{}\\ \texttt{/* think */ +memo(hungry)\}}\\ \texttt{shutdown()}\ \langle\texttt{failed(eating)}\rangle\end{array}\right]\texttt{\{ \}]\{ \}}\rangle^{\texttt{Phil.}}$$

(If the evaluation of the predicate `completed(eating)` was done before completion of predicate `eating` the result would have been `false`, and then its evaluation rescheduled.) Once the sub-activity `living` completes its execution, in the example of Fig. 1 it would be rescheduled (since its persistency condition is `true`).

4 PROPERTIES

We have defined a type system for FAL – not reported in the paper for lack of space. The soundness of the type system implies that the execution of well-typed agents and artifacts does not get stuck. The following properties of interaction between well-typed agents and artifacts, which are useful in concurrent programming with SIMPA, hold: (*i*) there is no `use` action specifying an operation control that is not part of the usage interface of the artifact; (*ii*) there is no `observe` action specifying an observable property that does not belong to the specified artifact; and (*iii*) an executing activity may be blocked only in a `sense` action over a sensor that does not contain the label specified in the filter—i.e., the agent explicitly stops only for synchronization purposes. Moreover, a type restriction on sensors – not present in the current type system – may be defined to enforce that there is no `sense` action indefinitely blocked on sensing event `e` due to the fact that the corresponding triggered operation was not designed to generate `e`.

5 RELATED WORK

The extension of the OO paradigm toward concurrency — i.e. object-oriented concurrent programming (OOCP) — has been (and indeed still is) one of the most important and challenging themes in the OO research. Accordingly, a quite large amount of theoretical results and approaches have been proposed since the beginning of the 80's, surveyed by works such as (Briot et al., 1998; Yonezawa and Tokoro, 1986; Agha et al., 1993; Philippsen, 2000). We refer to (Ricci et al., 2008) for a comparison of the agent and artifact programming model with *active objects* (Lavender and Schmidt, 1996) and *actors* (Agha, 1986) and with more recent approaches extending OO with concurrency abstractions, namely POLYPHONIC C# (Benton et al., 2004) and JOIN JAVA (Itzstein and Kearney, 2001) (both based on Join Calculus (Fournet and Gonthier, 1996)). Another recent proposal is STATEJ (Damiani et al., 2008), that proposes *state classes*, a construct for making the state of a concurrent object explicit. The objective of our approach is quite more extensive in a sense, because we introduce an abstraction layer which aims at providing an effective support for tackling not only synchronisation and coordination issues, but also the engineering of passive and active parts of the application, avoiding the direct use of low-level mechanisms such as threads.

6 CONCLUSIONS

We described FAL, a core calculus to provide a rigorous formal framework for designing agent-oriented languages and studying properties of agent-oriented programs. To authors knowledge, the only attempt that has been done so far applying OO formal modelling techniques like core calculi to study properties of agent-oriented programs and of agent-oriented extensions of object-oriented systems is (Ricci et al., 2008). A main limitation of the formalization proposed in (Ricci et al., 2008) is the lack of a type system that is able to guarantee well-formedness properties of programs. In this paper we formalized a larger set of features (including *agent agenda* and *artifact properties*) and provided a type soundness result.

The type system paves the way towards the analysis of the computational behaviour of agents. Properties that we are investigating mainly concerns the correct execution of activities, in particular: (i) there is no activity which are never executed because of their pre-condition; (ii) post-conditions for activity execution can be statically known, expressed as set of memos that must be part of the memo space as soon as the activity has completed; (iii) invariants for activity execution can be statically known, expressed as set of memos that must be part of the memo space while the activity is in execution; (iv) there is no internal action reading or removing memos that has not been previously inserted. We are investigating the suitable definition of pre/post/invariant conditions in terms of sets of memos that must be present or absent in the memo space, so that it would be possible to represent high-level properties related to set of activities, such as the fact that an activity A would be executed always after an activity A' or that an activity A and A' cannot be executed together. On the artifact side, the computational model of artifacts ensures a mutually exclusive access to artifact state by operations executed concurrently; more interesting properties could be stated by considering not only atomic but also structured operations, not dealt in this paper. We are also planning of integrating and comparing our approach based on static analysis with traditional verification techniques such as model-checking.

REFERENCES

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.

Agha, G., Wegner, P., and Yonezawa, A., editors (1993). *Research directions in concurrent object-oriented programming*. MIT Press, Cambridge, MA, USA.

Benton, N., Cardelli, L., and Fournet, C. (2004). Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804.

Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329.

Damiani, F., Giachino, E., Giannini, P., and Drossopoulou, S. (2008). A type safe state abstraction for coordination in java-like languages. *Acta Inf.*, 45(7-8):479–536.

Fournet, C. and Gonthier, G. (1996). The reflexive chemical abstract machine and the join calculus. In *POPL'96*, pages 372–385. ACM.

Igarashi, A., Pierce, B., and Wadler, P. (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450.

Itzstein, G. S. and Kearney, D. (2001). Join Java: an alternative concurrency semantics for Java. Technical Report ACRC-01-001, Univ. of South Australia.

Lavender, R. G. and Schmidt, D. C. (1996). Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Omicini, A., Ricci, A., and Viroli, M. (2009). Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 19. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

Philippsen, M. (2000). A Survey of Concurrent Object-Oriented Languages. *Concurrency Computat.: Pract. Exper.*, 12(10):917–980.

Ricci, A. and Viroli, M. (2007). SIMPA: An agent-oriented approach for prototyping concurrent applications on top of java. In *PPPJ'07*, pages 185–194. ACM.

Ricci, A., Viroli, M., and Cimadamore, M. (2008). Prototyping concurrent systems with agents and artifacts: Framework and core calculus. *Electron. Notes Theor. Comput. Sci.*, 194(4):111–132.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62.

Yonezawa, A. and Tokoro, M., editors (1986). *Object-oriented concurrent programming*. MIT Press, Cambridge, MA, USA.