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 (individual 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
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 formalising 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, 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

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.
ing 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-activities (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 syntactically 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 focusing (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); 
    } 
} 
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 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 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 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.
The expression `fail` models 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 1. Note that the types of parameters, in artifact operations and the type of fields and properties may not be sensors or 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 \( \gamma \) ranges over references to instance of agents, \( \alpha \) over artifacts, \( \sigma \) over sensors. **Configurations** are non-empty sets of agent/artifact/sensor instances.

**Sensor instances** are represented by \( \sigma = (I, \gamma, G, R) \), where \( \gamma \) is the instance identifier, and \( I \) is the queue of association labels/values representing the events generated (and not yet perceived) on the sensor. **Agent instances** are represented by \( \gamma = (I, \alpha, \gamma, G) \), where \( \gamma \) is the agent identifier, \( G \) is the type of the agent, \( I \) is the content of the `memo-space`, \( \alpha \) 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 \( I \) 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.

\[
R ::= a(\gamma)[Sr1 \ldots Srn] = e \mid \text{failed}\]

The name of the activity is \( a \), \( \gamma \) are the actual parameters of the current activity instance, \( Sr1 \ldots Srn \) 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 `failed` we say that activity \( a \) has `failed`. If the evaluation of a sub-activity is successful then it is removed from the

\[
\begin{align*}
U &::= G \mid A \mid C \\
T &::= U \mid Sns \\
GD &::= \text{agent } G \{ \text{Sns} \#; \text{Act} \} \\
Act &::= \text{activity } a(\overline{x}) \{ \text{agenda} (\overline{\text{SubAct}}) \{ e \} \} \\
SubAct &::= a(\overline{x}) : \text{pers} e : \text{pre } e \\
AD &::= \text{artifact } A \{ U \mid Up \mid \overline{Op} \} \\
Op &::= \text{operation } o(\overline{U} : \overline{\text{guard}}) \{ e \} \\
e &::= x \mid c \\
&\mid \text{spawn } G(\overline{x}) \mid \text{make } A(\overline{x}) \\
&\mid f = e \\
&\mid p = e \\
&\mid \text{signal}(l(e)) \\
&\mid \text{use } e.o(\gamma) : \text{sns } e \mid \text{filter } l \\
&\mid \text{focus}(e,e) \mid \text{unfocus}(e,e) \\
&\mid \text{observe } e.p \\
&\mid \text{?memo}(l) \mid \text{ +memo}(l(e)) \\
&\mid \text{memo}(l) \\
&\mid \text{started}(a) \mid \text{completed}(a) \mid \text{failed}(a) \\
&\mid \text{fail}
\end{align*}
\]

**Figure 3:** Syntax.
Artifact instances are represented by \( \alpha = (\mathcal{T} = \mathcal{V}, S, 0_1 \ldots 0_n)^a \) where \( a \) is the artifact identifier. \( \mathcal{T} \) the type of the artifact, the sequence of pairs \( \mathcal{V} \) associates a value to each field of \( a \). the sequence \( \mathcal{V} \) represents the sensors that agents focusing on \( a \) are using, and \( 0_n, 1 \leq i \leq n, \) are the operations that are in execution. We consider \( 0_n \) a queue with first element \( Q_1 \) and last \( Q_2 \). For simplicity, we do not consider steps in this paper, although we have a full formalization including them. Artifacts are single threaded (differently from agents that may have more activity running at the same time) only the operation \( 0_n \) is being evaluated. A running operation, \( 0_n \), is defined as follows.

\[
0_n := (\sigma, o(e){e'})
\]

where \( \sigma \) identifies the sensor associated with the operation which was specified by the agent containing the \( e \) that started the operation, and that is used to collect events generated during the execution of the operation by \( e \). If the expression \( (e) \) is different from \( e \) 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:

\[
\gamma_{main} = (\emptyset, \emptyset) \begin{array}{l}
\text{spawn Philosopher}(0,1,\alpha) \\
\text{spawn Philosopher}(4,0,\alpha)
\end{array}
\]

The five spawn expressions are evaluated from left to right. The evaluation of the expressions \( \text{spawn Philosopher}(0,1,\alpha) \) reduces to \( \gamma_0 \) and adds to the configuration the agent instance

\[
(1) \gamma_0 = (\emptyset, \emptyset) \begin{array}{l}
\text{prepare}() (\text{true}) \\
\text{living}(0,1,\alpha) (\text{memo}(\text{hungry}))
\end{array}
\]

and the sensor instance \( \sigma_0 = (\emptyset)^{\text{Sns}} \).

Similarly, the reduction of the other spawn expressions generates four agent instances and four sensor instances producing the configuration:

\[
\gamma_{main} = (\emptyset) \ \sigma_0 = (\emptyset) \ldots \sigma_4 = (\emptyset) \alpha = (\ldots) \gamma_1 = (\ldots) \cdot \gamma_n = (\ldots)
\]

in which the agent \( \gamma_{main} \) is inactive, having finished the evaluation of its body. The artifact \( \alpha \) 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 \( \text{prepare} \) and \( \text{living} \)). Our modeling make use of nondeterministic evaluation rules, but parallel execution could be modeled.

Continuing from (2) the precondition \( \text{living}(0,1,\alpha) (\text{false}) \) would have been replaced with \( \text{living}(0,1,\alpha) (\text{memo}(\text{hungry})) \). Continuing from (2) the precondition \( \text{meme}(\text{hungry}) \) of \( \text{living} \) evaluates to true and the sub-activity \( \text{living}(0,1,\alpha) (\text{true}) \) is replaced by the corresponding run-time activity resulting in the following:

\[
\gamma_n = (\emptyset, \sigma_0, \emptyset) \begin{array}{l}
\text{prepare}() (\text{true}) \\
\text{living}(0,1,\alpha) (\text{false})
\end{array}
\]

Next time the sub-activity \( \text{living} \) was scheduled for execution \( \text{living}(0,1,\alpha) (\text{false}) \) would have been replaced with \( \text{living}(0,1,\alpha) (\text{memo}(\text{hungry})) \).
the following

\[
\begin{align*}
\text{(3) eating} & : 0,1, \alpha(1) \\
& \left\{ \begin{array}{l}
\text{use } \alpha \text{.getForks}(0,1) : \text{sns } \sigma_0 \\
\text{releaseForks(acquired)}; \text{ /* eat */} \\
\text{use } \alpha \text{.releaseForks}(0,1); \\
\text{-memo(hungry)}
\end{array} \right.
\end{align*}
\]

(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 \( \alpha \text{.getForks}(0,1) : \text{sns } \sigma_0 \) that schedules the operation getForks in the artifact instance \( \alpha \) yielding

\[
\alpha = (\text{.isBF} = \{t, f, f, f, f\} : 0, 0, (\sigma_0, \text{getForks } (\{e_1\}) (\{e_2\})))^{\text{false}}
\]

where \( e_0^{(1)} \) is \( \text{not}(\{\text{.isBF}(0)\}) \) and \( \text{not}(\{\text{.isBF}(1)\}) \)

and \( e_0^{(1)} \) is \( \text{.isBF}(0) = \text{true}; \text{.isBF}(1) = \text{true}; \text{signal(forks acquired)} \).

The guard \( e_0^{(1)} \) reduces to true. The reduction of \( \sigma_0 \) updates the array \( t \) to \( \{t, t, f, f, t\} \) and adds the label forks acquired to the queue of events of the sensor instance \( \sigma_0 \), yielding \( \sigma_0 = \{\text{forks acquired}\}^{\text{true}} \).

Other agents may schedule the operation the artifact \( \alpha \) yielding the following

\[
\alpha = (\text{.isBF} = \{t, t, f, f, t\} : 0, 0, (\sigma_0, \text{getForks } (\{e_1\}) (\{e_2\})))^{\text{true}}
\]

So the guard \( e_1^{(1)} \) \( \{\text{not}(\{\text{.isBF}(1)\}) \) and \( \text{not}(\{\text{.isBF}(2)\}) \) \)

would evaluate to false, and the associated operation would be rescheduled and put at the rear of the queue yielding the following

\[
\alpha = (\text{.isBF} = \{t, t, f, f, t\} : 0, 0, (\sigma_0, \text{getForks } (\{e_1\}) (\{e_2\})))^{\text{true}}
\]

so the evaluation of the guard of the getForks operation invoked by \( \gamma \) may start and will successfully acquire the forks for \( \gamma \). At the same time, the expression \( \text{sense } \sigma_0 : \text{filter forks acquired} \) in (3) could be evaluated, receiving the event forks acquired and removing it from the sensor instance \( \sigma_0 \) which becomes \( \sigma_0 = \{0\}^{\text{true}} \).

The code "/* eat */" may be executed and, at the end of its execution the expression \( \text{use } \alpha \text{.releaseForks}(0,1) \) schedules the operation releaseForks on the artifact \( \alpha \) and then \( \text{-memo(hungry)} \) 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 \( \gamma_0 \) to be:

\[
\{0, \sigma_0, \text{main}(living(0,1), \text{thinking}()) | \{t, t, f, f, t\} / \text{ /* eat */} / \text{-memo(hungry)} / \text{shutdown}(/ \text{ failed(eating)}) \}^{\text{true}}
\]

(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 (OOPC) — 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 synchronization and
coordination issues, but also the engineering of passive and active parts of the application, avoiding the direct use of low-level mechanisms such as threads.

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

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