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*  
damiani@di.unito.it

Paola Giannini  
*Dipartimento di Informatica, Università del Piemonte Orientale, Via Bellini 25/G, 15100 Alessandria, Italy*  
giannini@mfn.unipm.it

Alessandro Ricci, Mirko Viroli  
*DEIS, Università di Bologna, Via Venezia 52, 47023 Cesena, Italy*  
{a.ricci, mirko.viroli}@unibo.it

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 architec-
ture, 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:

```
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(4,5,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

+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.

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, specify-
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.
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 activity, which is used by two User agents. After execution of the action, the sensor 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 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, persistence predicates and guards. We use the overbar se-
Each agent/artifact/sensor instance has a unique reference identity, provided by a transform sets of instances of agents/artifacts/sensors. The set of reduction rules that define fields and properties are accessed/modified. Activities mentioned in todo lists are defined, and only right context (artifact or agent), operation used, and state enforcing the fact that expressions occur in the right context (artifact or agent), operation used, and state of execution of the sub-activities. 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.

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 \( l \) ranges over labels.

The expression \( \text{fail} \) model failures in activities, such as the evaluation of \( ?\text{memo}(l) \) and \( \neg \text{memo}(l) \) 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, \( \text{signal}(l(e)) \), 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 \( \sigma = (\overline{I}v)^{\text{sns}} \), where \( \sigma \) is the instance identifier, and \( \overline{I}v \) is the queue of association labels/values representing the events generated (and not yet perceived) on the sensor.

Agent instances are represented by \( \gamma = (\overline{I}v, \overline{\sigma}, R)^{\alpha} \), where \( \gamma \) is the agent identifier, \( G \) is the type of the agent, \( \overline{I}v \) 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, \( \text{main} \), that was started when the agent was created. The sensor instances in \( \overline{I}v \) are in one-to-one correspondence with the sensor variables declared in the agent and are needed since every agent uses its 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.

\[
R ::\ a(\overline{v})[S_{r_1} \ldots S_{r_n}][\{e\}] \mid \text{failed}^{a}
\]

The name of the activity is \( a \), \( \overline{v} \) are the actual parameters of the current activity instance, \( S_{r_1} \ldots S_{r_n} \) 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.)
we say that activity $a$ has failed. If the evaluation of a sub-activity is successful then it is removed from the set $Sr_1 \cdots Sr_n$. 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, $s(\tau)(e)$ where $e$ is different from true or false (it is the state of evaluation of the precondition) when $e = true$, the term $s(\tau)(true)$ is replaced with the initial state of the evaluation of the activity $a$ with parameters $\tau$. When $e = false$ the evaluation of the precondition of $a$ is rescheduled. Therefore:

$$Sr := a(\tau)(e) \mid R$$

Artifact instances are represented by $\alpha = (\tau = \mathcal{F} = \mathcal{V}, \mathcal{P} = \mathcal{F} \mathcal{G} \mathcal{A} \mathcal{G} \mathcal{S} \mathcal{O} \mathcal{O} \mathcal{O} \mathcal{O}, \mathcal{O} \mathcal{O} \mathcal{O} \mathcal{O})$ where $\alpha$ is the artifact identifier, $A$ the type of the artifact, the sequence of pairs $\mathcal{F} \mathcal{G} \mathcal{A} \mathcal{G} \mathcal{S} \mathcal{O}$ associates a value to each field of $A$, the sequence of pairs $\mathcal{P} \mathcal{G} \mathcal{A} \mathcal{G} \mathcal{S} \mathcal{O}$ associates a value to each property of $A$, the sequence $\mathcal{G}$ represents the sensors that agents focusing on $A$ are using, and, $0, 1 \leq i \leq n$, are the operations that are in execution. We consider $0, 1 \cdots O_n$ a queue with first element $O_0$ and last $O_1$. (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 we have a full formalization including them.) Artifacts, we do not consider steps in this paper, although we have a full formalization including them.) Artifacts, single threaded and (differently from agents that may have more activity running at the same time) only the operation $O_n$ is being evaluated.

A running operation, 0, is defined as follows.

$$0 := (\sigma, o(e)\{e\})$$

where $\sigma$ 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 $\mathcal{V}$. If the expression $o(e)$ 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:

$$\alpha = \{.main = \{Table=\{make Table(new Bool[5]);\};
\text{spawn Philosopher}(0,1,t);\text{spawn Philosopher}(4,0,t)\}\}$$

The expression $new\ \text{Bool}[5]$ reduces to the array $\{t,t,f,f,f\}$ (in the array we use $f$ for false and $t$ for true). Then the expression make

Table($\{t,t,f,f,f\}$) reduces to an artifact reference $\alpha$ and adds to the configuration the initial artifact instance that follows:

$\alpha = \{.main = \{Table=\{make Table(new Bool[5]);\};
\text{spawn Philosopher}(0,1,t);\text{spawn Philosopher}(4,0,t)\}\}$

After the initialization of the local variable $\tau$ the agent instance $\gamma_{main}$ becomes

$$\gamma_{main} = \{.main = \{\text{spawn Philosopher}(0,1,\alpha)\} \}$$

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

$$(1) \gamma_0 = \{.c_{main} = \{\text{prepare}(\alpha)\} \}$$

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

$$\gamma_{main} = \{.main = \{\text{spawn Philosopher}(0,1,\alpha)\} \}$$

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.

Going back to (1), since the precondition of the runtime sub-activity $\text{prepare}(\alpha)$ of the activity main of the agent $\gamma_0$ is true the expression $\text{prepare}(\alpha)(true)$ is replaced by $\text{prepare}(\alpha)(\{\text{living}(0,1)\} \{\text{memo(hungry)}\})$ (whose evaluation causes the insertion of the label hungry into the memo of $\gamma_0$) and then since the body is fully evaluated prepare is removed from the sub-activities of main, yielding

$$\gamma_0 = \{.c_{main} = \{\text{living}(0,1,\alpha) \{\text{memo(hungry)}\}\} \}$$

If instead of evaluating the sub-activity $\text{prepare}$ we would have evaluated the precondition of the sub-activity $\text{living}$, the result would have been having

$$\gamma_0 = \{.c_{main} = \{\text{prepare}(\alpha)(true),\text{living}(0,1,\alpha)(false)\} \}$$

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

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

$$\gamma_0 = \{.c_{main} = \{\text{prepare}(\alpha)(true),\text{living}(0,1,\alpha)(false)\}\}$$

1The syntax of FAL does not include local variables and array object values. In this example, we will handle the local variable $\tau$ by replacing, after its declaration/initialization, all its occurrences with its value.
The precondition \text{memo}\{\text{hungry}\} evaluates to \text{true} and the sub-activity \text{eating}(0,1,\alpha)(\text{true}) is replaced by the corresponding run-time activity resulting in the following

\begin{equation}
\begin{array}{l}
\quad \quad \text{use } \alpha.\text{getForks}(0,1) \quad \text{sns } \sigma_0 \\
\quad \quad \quad \text{sense } \sigma_0.\text{filter } \text{forks}\_\text{acquired} \\
\quad \quad \quad \quad \text{\quad /* eat */} \\
\quad \quad \quad \quad \text{use } \alpha.\text{releaseForks}(0,1) \\
\quad \quad \quad \quad \text{-memo } \text{\{hungry\}} \\
\quad \quad \\}
\end{array}
\end{equation}

(Note that both \text{completed(eating)} and \text{failed(eating)} would evaluate to \text{false}.) The evaluation of the body of the \text{activity} can now start by reducing the expression \text{use } \alpha.\text{getForks}(0,1) \quad \text{sns } \sigma_0, that schedules the operation \text{getForks} in the artifact instance \alpha yielding

\begin{equation}
\alpha = (\{\text{isBF} = \{t,t,f,f,f\} \bullet \text{true}; \text{isBF}[0] = \text{true}; \text{isBF}[1] = \text{true}; \text{signal } \text{forks}_{\text{acquired}} \})^{\text{false}}
\end{equation}

where \text{\alpha}_0' is \text{(not }\{\text{isBF}[0]\} \text{ and } \text{(not }\{\text{isBF}[1]\}))\text{true} and \text{\alpha}_0 \text{ is } \{\text{isBF} = \text{true}; \text{isBF}[1] = \text{true}; \text{signal } \text{forks}_{\text{acquired}}\}. The guard \text{\alpha}_0' \text{ reduces to } \text{true}. The reduction of \text{\alpha}_0 updates the array \text{t} to \{t, t, f, f, f\} and adds the label \text{forks}_{\text{acquired}} to the queue of events of the sensor instance \sigma_0, yielding \text{\alpha}_0 = (\text{forks}_{\text{acquired}})^{\text{false}}\text{sns}. Other agents may schedule operation the artifact \alpha. For instance, if the agent \gamma_1 and \gamma_2 invoke the operation \text{getForks} on \alpha, when the evaluation of \text{getForks} for the agent \gamma_0 was completed the state of the artifact would be

\begin{equation}
\alpha = (\{\text{isBF} = \{t,t,f,f,f\} \bullet \text{false}, \sigma_0.\text{getForks } \{\text{\alpha}_0'\} (\text{\alpha}_0)\})^{\text{false}}
\end{equation}

So the guard \text{\alpha}_1 (\text{(not }\{\text{isBF}[1]\} \text{ and } \text{(not }\{\text{isBF}[2]\})) \}) \text{ would evaluate to } \text{false}, and the associated operation would be rescheduled and put at the rear of the queue yielding the following

\begin{equation}
\alpha = (\{\text{isBF} = \{t,t,f,f,f\} \bullet \text{false}, \sigma_1.\text{getForks } \{\text{\alpha}_1'\} (\text{\alpha}_1)\})^{\text{false}}
\end{equation}

so the evaluation of the guard of the \text{getForks} operation invoked by \gamma_2 may start and will successfully acquire the forks for \gamma_2. At the same time, the expression \text{sense } \sigma_0 : \{\text{filter } \text{forks}\_\text{acquired} \text{ in } (3) \text{ could be evaluated, perceiving the event } \text{forks}_{\text{acquired}} \text{ and removing it from the sensor instance } \sigma_0 \text{ which becomes } \sigma_0 = (\emptyset)^{\text{false}}\text{sns}. The code "/* eat */" may be executed and, at the end of its execution the expression \text{use } \alpha.\text{releaseForks}(0,1) \quad \text{schedules the operation } \text{releaseForks} \quad \text{on the artifact } \alpha \text{ and then } \text{-memo } \{\text{hungry}\} \quad \text{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 \text{completed(eating)} becomes true and the sub-activity thinking could be executed resulting in \gamma_0 to be:

\begin{equation}
\{\emptyset, \sigma_0, \text{main}\{\text{living}(0,1,\alpha) \quad \text{thinking}(|1|)\quad \text{\{/* think */ } \text{+memo } \{\text{hungry}\} \quad \text{•shutdown}(|\text{failed(eating)}) \}}\text{\}}^{\text{false}}
\end{equation}

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

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