Scientific Annals of Computer Science vol. 22 (2), 2012, pp. 267–326
doi: 10.7561/SACS.2012.2.267

Standard Type Soundness for Agents and Artifacts

Ferruccio DAMIANI[1], Paola GIANNINI[2], Alessandro RICCI[3], Mirko VIROLI[4]

Abstract

Formal models, core calculi, and type systems, are important tools for rigorously stating the more subtle details of a language, as well as characterizing and studying its features and the correctness properties of its programs. In this paper we present FAAL (Featherweight Agent and Artifact Language), a formal calculus modelling the agent and artifact program abstractions provided by the simpA agent framework. The formalisation is largely inspired by Featherweight Java. It is based on reduction rules applied in certain evaluation contexts, properly adapted to the concurrency nature of simpA. On top of this calculus we introduce a standard type system and prove its soundness, so as to guarantee that the execution of a well-typed program does not get stuck. Namely, all primitive mechanisms of agents (activity execution), artifacts (field/property access and step execution), and their interaction (observation and invocation) are guaranteed to be used in a way that is structurally compliant with the corresponding definitions: hence, there will not be run-time errors due to FAAL distinctive primitives.

Keywords: multi-agent systems, concurrency, type systems

A preliminary version of some of the results presented in this paper appeared in previous work by the same authors [17, 18]. Partially supported by MIUR PRIN 2008 DISCO project. The funding bodies are not responsible for any use that might be made of the results presented here.

[1]Università di Torino, Dipartimento di Informatica, Corso Svizzera 185, I-10149 Torino, Italy. Email: ferruccio.damiani@unito.it

[2] Università del Piemonte Orientale, Dipartimento di Informatica, Via Teresa Michel 11, 15121 Alessandria, Italy. Email: giannini@di.unipmn.it

[3]Alma Mater Studiorum, Università di Bologna, DEIS, via Venezia 52, 47521 Cesena, Italy. Email: a.ricci@unibo.it

[4]Alma Mater Studiorum, Università di Bologna, DEIS, via Venezia 52, 47521 Cesena, Italy. Email: mirko.viroli@unibo.it

1 Introduction

Concurrency is gaining momentum in the development of software systems, given the widespread diffusion of multi-core architectures and the Internet. As nicely put forward by Sutter and Larus in [58]: "*the free lunch is over*", which means that the capability of writing well-engineered concurrent programs – efficiently and effectively exploiting the available hardware parallel features – is nowadays a requirement of any mainstream language for programming-in-the-large, such as current OO programming languages. As such, it becomes more and more important to enhance programming languages with proper high-level abstractions that can "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [58].

From that perspective, a large amount of proposals have been developed in the literature since the 80's, many extending existing sequential programming paradigms with concurrency features, such as the case of object-oriented concurrent programming approaches [10, 63, 3]. Among the others, two main prominent families of approaches are *actors* [2] and *agents* [37]. The *actor* computing model is the reference model of several concurrent programming languages and frameworks [39, 40], including Erlang [4], Scala [26], and ActorFoundry [42]. An actor-based program is defined by a set of autonomous active entities that communicate solely by asynchronous message passing. On the other hand, *agents* and *multi-agent systems* – having their roots in the context of Distributed Artificial Intelligence [61, 54] – are nowadays a main paradigm for engineering complex software systems exhibiting features of concurrency, distribution/decentralisation, autonomy, and flexibility [37]. They gave rise to a number of agent oriented programming languages and frameworks—surveyed in [8, 7]. Similarly to actors, agents are active entities communicating by asynchronous message passing. Differently from actors, in agent systems an explicit notion of *environment* is considered that defines agent *situatedness*: agents are reactive systems that continuously perceive their environment and autonomously decide which actions to perform, according to their designed objective [60]. Then, the environment can be exploited also as a first-class abstraction to realise indirect/mediated forms of communication besides message passing [59].

Independently of the abstractions we use, a main problem with the engineering of concurrent programs lays in the difficulty of assessing their correctness. Formal models based on core calculi and associated type system are a widely accepted and standard approach for rigorously stating the

more subtle details of a programming language, characterizing and studying its features and the correctness properties of its programs. A prominent example for this approach is rooted in the Featherweight Java calculus (FJ) [34], which has being used as the basis for studying the properties of a number of extensions of OO languages including sophisticated type systems [35]. Besides the formalisation of sequential programming languages, core calculi can be valuable tools also for rigorously studying concurrency models, concurrent programming languages, or programming languages providing some support for concurrent programming.

Therefore, along with the design of new abstractions, a main research issue in concurrency programming is the definition of proper formal models, calculi, and type systems, analogously to the sequential case. The main examples in this direction are CAP [12], a process calculus based on the actor model, Honda and Tokoro's object calculus with asynchronous communication [31], and the join calculus [24].

The same situation does not hold for agent programming languages, which are the focus of this paper. There are many papers on the formal semantics of high-level agent programming languages (e.g., [29, 22, 46, 57, 21]), and in particular on their operational semantics; none of them, however, deals with aspects such as typing and type safety. The main reason is that most of them are logic-based, and have been introduced for solving problems in the Distributed Artificial Intelligence context; hence, existing formalisations focussed on aspects related to the underlying logic-based model of the languages, in order to rigorously define the reasoning capabilities of agents. A prominent example is [43], where the situation calculus is used to formally specify the behaviour of cognitive agents. On the other hand, some agent-oriented programming frameworks have been proposed as general-purpose approaches to develop concurrent and distributed systems, as an extension of OOP. Main examples are Jade [5] and simpA [53]. These approaches would clearly benefit from a proper formalisation including a type system capturing the essential features of the agent-oriented abstractions on which they are based.

Accordingly, in this paper we introduce a calculus with type system called FAAL (Featherweight Agent and Artifact Language), capturing the relevant and distinctive features of simpA and the A&A (Agents and Artifacts) conceptual model [47], introduced in the context of Agent Oriented Software Engineering. A FAAL program is given by a set of autonomous agents that work concurrently inside a shared environment mod-

ularised in terms of non-autonomous entities called *artifacts*. A distinctive aspect of this language is that it is largely inspired by FJ (it is based on reduction rules applied in certain evaluation contexts), but it also includes a number of techniques used in the context of concurrent models like process algebras—e.g., the use of parallel composition of components, and synchronisation of agent-artifact behaviour. A (well-formed) 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 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.

On top of this calculus we define a type system that ensures the correct evolution of a system, through the standard properties of progress, and preservation of well-formed configurations. In particular, the type soundness result we prove guarantees that all primitive mechanisms of agents (activity execution), artifacts (field/property access and step execution), and their interaction (observation and invocation) are used in a way that is structurally compliant with the corresponding definitions. As in most strongly type languages, this helps the programmer in writing provenly correct programs in a modular way – a rather necessary mean for building complex applications – and paves the way for a more extensive behavioural analysis.

Organisation of the paper Section 2 introduces the basic concepts underlying FAAL – and SIMPA – programming model. Section 3 presents the (typed) syntax, and the operational semantics of the FAAL language. In Section 4 we state the theorems related to type soundness. Section 5 revises the related papers and Section 6 outlines possible directions for further work. The appendix contains the proofs of the theorems of Section 4.

2 FAAL Agent-Oriented Abstractions and Type System: An Informal Overview

This section provides a brief informal description of the basic abstractions on which the FAAL calculus is based. The interested reader can find a more detailed account of the programming model in the context of the SIMPA framework [53].

A program in FAAL is represented by a (dynamic) set of autonomous entities called *agents*. Agents work concurrently inside a shared environment represented by a (dynamic) set of non-autonomous entities called *artifacts*.

Agents and *artifacts* are the basic high-level and coarse-grained abstractions available in the A&A conceptual 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 [47]. Agents are used to model task-oriented components of a system, and are autonomous in the sense that they encapsulate the logic and control of task execution. Artifacts model instead function-oriented components of a system, used by agents for accomplishing their individual and collective tasks.

In the remainder of this section we will introduce the essential elements of *(i)* agents, *(ii)* artifacts and *(iii)* their interaction in FAAL by using a simple example, given by four kinds of agents (`Main`, `Init`, `User`, `Observer`) and an artifact (`Counter`). The `Main` and `Init` agents set up the system, while two `User` agents and an `Observer` agents work cooperatively by using the shared artifact, respectively by incrementing it (`Users`) and reacting to its changes (`Observer`).

Agent Abstraction An agent in FAAL is a stateful entity whose job is to pro-actively execute a structured set of *activities* as specified by the agent programmer. Such activities (which may possible be non-terminating) finally result in executing sequences of atomically-executed *actions*: internal actions that inspect/change agent state, or external actions by which interaction with the agent environment is achieved. Examples of agents with a single `main` activity are given by the `Main` and `Init` agent:

```
agent Main {
    activity main() {
       spawn Init(make Counter(0));
    }
}

agent Init {
    activity main(Counter c) {
       spawn Observer(c);
       spawn User(c);
       spawn User(c);
    }
}
```

Built-in actions `make` and `spawn` are used create artifacts and to spawn agents. 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 and associatively read/retrieve chunks of information called *memos*. A memo is a tuple, characterised by a label and an ordered set of arguments. Besides keeping track of state information, memos are useful to coordinate the execution of structured activities, as will be shown in the following.

A basic set of internal actions is available to agents that work 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. Labels of memos have a type, which constrains the type of values used in arguments.

The computational behaviour of an agent can be defined as a hierarchy of activities (corresponding to the execution of some tasks). An example is given by the `Observer` agent:

```
agent Observer {
    activity main(Counter c) :agenda (
                prepare(c)  :pre tr
                            :pers fls,
            monitoring(c)   :pre completed(prepare)
                            :pers (not memo(finished))
    ) { }
    activity prepare(Counter c) { ... }
    activity monitoring(Counter c) { ... }
}
```

Activities can be simple or structured, and are represented by `activity` blocks, providing the name of the activity, parameters, and behaviour specification. The behaviour of a simple activity (`prepare` and `monitoring` in the above example) is composed of a flat sequence of actions (inside curly brackets), which specifies a single control flow.

For structured activities (like `main` in the above example), an agenda is specified, containing a set of sub-activities, called *todos*. These sub-activities, which run concurrently, have to be completed before the activity may start the execution of its sequence of actions (inside curly brackets). This leads to a hierarchical structure of running activities.

A todo contains the name of the sub-activity to be executed, a precondition over the inner state of the agent that must hold for the specified sub-activity to start (clause `:pre`), and a persistence attribute related to the sub-activity execution (clause `:pers`).

Preconditions are expressed as a boolean expression over a basic set of predefined predicates. 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 `Observer` example, in the structured activity `main`, the activity `monitoring` is executed when the activity `prepare` completed — `completed(A)` is a built-in predicate which is true is the specified activity `A` has completed.

When the precondition of a todo item holds (for an activity in execution listing such a 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 synchronised by properly specifying preconditions in todos, hence in a declarative way. If a todo is declared persistent, when the sub-activity is completed the todo is re-inserted into the agenda. The persistence attribute can also specify the condition under which the activity should persist. For instance, the todo item about the `monitoring` activity is declared persistent until a `finished` memo is found.

The type system of FAAL, in addition to the standard checks for expressions, will also ensure that activities mentioned in todo lists and in preconditions/persistence predicates, are those defined for the agent. This is the key to guaranteeing that the only form of agent-to-agent interaction is via artifacts, as dictated by the A&A conceptual model.

Artifact Abstraction An artifact is composed of 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 of set of state variables analogous to private instance fields of objects; and *(iii) operations*, which embody the computational behaviour of artifacts. A minimal example of artifact is given by the following `Counter`:

```
artifact Counter {
    obsprop int count;
    Counter(int c) { .count = c; }
    operation inc() { .count = .count+1; }
}
```

This artifact has one observable property (`count`), no inner state variable, and one operation (`inc`) to increment the counter.

Both state variables and observable properties are declared similarly to instance fields in objects; observable properties are prefixed by `obsprop` keyword. In both cases, a dot notation (e.g. `.count`) is used both for l-values and r-values, to syntactically distinguish them from parameters. Operations can be defined by method-like blocks, prefixed by the keyword `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.

Operations can be either *atomic*, executed as a single computational step, or *structured*, i.e. composed of multiple atomic operation steps. For each operation a *guard* can be specified (`:guard` declaration), representing the condition that must hold for scheduling the execution of the operation code. Structured operations and guards are essential in easily implementing *coordination artifacts*, i.e. artifacts that are designed to mediate agent interaction and provide some coordination functionality. A simple example is given by a *synchronisation barrier* artifact:

```
artifact Barrier {
    int v;
    Barrier(int n){ .v = n; }
    operation synch() { .v = .v-1; step allSynched() }
    step allSynched() :guard (.v == 0) {}
}
```

This artifact is useful for synchronizing a set of `n` agents, each executing the `synch` operation as a synchronisation point. The hidden state variable `v` keeps track of the number of agents that executed `synch` so far. The `synch` operation is composed of two steps: in the first (implicit) one, the internal variable is decremented; the second one is executed only when `v` reached zero, meaning that all agents reached the synchronisation point. Other examples of components that can be suitably modelled as coordination artifacts are bounded buffers, blackboards, communication channels [53].

To be useful, an artifact should typically provide some level of *observability*. This is achieved both by generating observable events through the `signal` primitive, and by defining observable behaviours. The events generated through the `signal` primitive can be sensed by the agent using the artifact — i.e. by the agent which started the operation. An observable event is represented by a labelled tuple, whose label represents the kind of

the event and the information content. The end of an operation is an observable behaviour generating the event `op_exec_completed`, whereas when a property changes, an event of type `prop_updated` is generated (with the new value of the property as a content). For instance, in the `Counter` artifact `inc` operation generates a `prop_updated(count, Value)` event each time the observable property `count` is updated. These events may be sensed by all the agents that are *focussing* (observing) the artifact (more details below).

Type checking an artifact is very similar to typechecking for classes, as fields and properties are typed. Our type system is nominal (like that of Java), so the type of an artifact is the name used in its definition.

Agent-Artifact Interaction Model We explain the details of agent-artifact interactions based on the full program of the example reported in Figure 1.

As already stated, artifact *use* and *observation* are the basic form of interaction between agents and artifacts. The use of an artifact 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. In the abstract language presented here, sensors used by an agent are declared at the beginning of the `agent` block (see sensor `s` in agent `Observer`)—note that each agent instance gets its private instance of the sensors.

In order to trigger operation execution, the `use` action is provided (exemplified in activity `usingCount` of agent `User`), specifying the target artifact (`c`), the operation to execute (`inc`) and optionally the identifier of the sensor (`s`) used to collect observable events generated by the artifact. The type system checks that agents invoke only operations that are defined for the artifact type, and that sensors are defined for the agent. It is important to note that no implicit control coupling exists between an agent and an artifact while an operation is executed.

The sensor that one may provide with the `use` primitive is associated with the activation of the operation, so that a synchronisation may be created between the agent and the artifact. During the execution of the operation the artifact may perform a `signal`, which adds to the associated agent sensor a perception (represented by a pair of a label and a value) that may be sensed by the agent via a `sense` primitive. The execution of a sense is blocked until there is a perception available, in which case it is returned. The code of operation `inc2` in the `Counter` artifact shows an example of signal,

```
artifact Counter {
    obsprop int count;
    Counter(int c) { .count = c; }
    operation inc() {
       .count = .count+1;
    }
    operation inc2() {       // variant showing signalling
       .count = .count+1;
       signal(val(.count));
    }
}

agent Main {
    activity main() { spawn Init(make Counter(0)); }
}

agent Init {
    activity main(Counter c) { spawn Observer(c); spawn User(c); spawn User(c); }
}

agent Observer {
    Sns s;
    activity main(Counter c) :agenda (
               prepare(c)  :pre tr
                           :pers fls,
            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 {
    Sns s;
    activity main(Counter c)  :agenda ( usingCount(c) :pre tr  :pers tr ) { }
    activity usingCount(Counter c) {
             use c.inc() :sns .s;
    }
    activity usingCount2(Counter c) { // variant showing sensing
             use c.inc2() :sns .s;
             ... // code that does not need the updated counter
             sense .s :filter val;
             ... // code that may assume that the counter has been updated
    }
}
```

Figure 1: The complete program including an `Observer` agent continuously observing a `Counter` Artifact, which is concurrently used by two `User` agents.

occurring each time the counter is updated and using `val` as label and the counter status `.count` as value. On the other side, if the agent wants to know whether the counter has been updated it should specify a sensor along with the `use` primitive, and use a `sense` when it needs the value, as shown in `usingCount2` activity of `User` agent. Note that a filter `val` is specified at sensing time, to select only the events labelled `val`—SIMPA provides general filters based on regular-expression patterns, matched over the event type (a string), while in FAAL we model a simple matching on label names. In general, sensing signals is the mechanism by which agents can get output results that could be possibly generated by the operation execution.

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 artifact – 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 is used to stop observing the artifact. In the example, the `Observer` agent executes a `focus` on the artifact in the `prepare` activity.

In our formalisation we shall model this kind of observation by restricting the observation to the completion of operations of an artifact, and the updating of its properties. To this end, we used the Observable/Observed pattern: agents insert sensors in the run-time artifacts to be observed. These sensors are used by the artifact to signal an observable event, and are sensed by the observing agent. Our type system ensures that sensors can only be defined in agents, and may not be passed as parameters to operations, so that artifacts cannot be aware of which sensor they are signalling to. This makes it possible to enforce a discipline of programming which is faithful to the A&A conceptual model, where artifacts as observable entities are not required to keep track of who is observing or using them: this is part of the interaction model and the artifact programmer can (and should) design artifact structure and behaviour abstracting from such a detail.

A main objective of FAAL is to have a formal framework to study errors of different kinds that can occur when executing SIMPA agent-oriented programs, having defined a sound notion of type of agent-oriented abstractions. Almost all existing agent programming languages are untyped or weakly typed; therefore, many simple but important types of errors are discovered only by executing a program [51], at runtime. The same holds

also for the SIMPA framework, which is based on pure Java types. Then, it is not possible – for instance – to detect at compile time if an agent is specifying a wrong operation when using an artifact, or if an agent tries to sense an event which is never generated by the focused artifact. In FAAL these simple errors can be clearly detected statically by having defined a suitable type system for agent and artifact abstractions.

3 FAAL Syntax, Typing and Operational Semantics

3.1 Syntax

The syntax of FAAL (FEATHERWEIGHT AGENT AND ARTIFACT LANGUAGE) is presented in Figure 2.

In presenting the calculus we use a set of conventions for names: `G` ranges over agent names, `A` ranges over artifact names, and `C` denotes the types of basic values (including `Bool` and `Unit`). For program identifiers: `s` denotes sensor names, `l` labels, `f` field names, and `p` property names. Finally we will use `a` for activity names, and `o` for operations and steps.

The overbar sequence notation is used according to [34]. For instance: "$\overline{\texttt{f}}$" denotes the possibly empty sequence "$\texttt{f}_1, ..., \texttt{f}_n$", the pair "$\overline{\texttt{U}}\,\overline{\texttt{x}}$" stands for "$\texttt{U}_1\,\texttt{x}_1, ...,\ \texttt{U}_n\,\texttt{x}_n$", and "$\overline{\texttt{U}}\,\overline{\texttt{f}};$" stands for "$\texttt{U}_1\,\texttt{f}_1; ...; \texttt{U}_n\,\texttt{f}_n;$". The empty sequence is denoted by "$\emptyset$".

There are very minor differences between the syntax of the calculus and that of the language used for the examples, namely, tuples are not first-class but are seen as specific kinds of basic values, and specifiers (`:agenda`, `:pers`, `:pre`, `:guard` and `:sns`) are mandatory instead of optional.

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 `l`. Note that the types of parameters in artifact operations and the type of fields and properties may not be sensors. Moreover, the signal expression, `signal(l(e))`, does not specify a sensor. Therefore, sensors may not be explicitly manipulated by artifacts.

As already hinted in the overview, operations can be either executed as a single computational step, or be composed of multiple atomic operation steps. Operation steps are implemented by operation-like blocks qualified with `step`, and can be triggered (enabled) by means of the `next` primitive specifying the name of the step to be enabled next and possibly its parameters.

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

Figure 2: Syntax

3.2 Typing

The FAAL calculus is equipped with a small-step operational semantics (cf. Section 3.3). This section presents the type system of FAAL. The main property enforced by the type system is the fact that the execution of a well-typed program does not get stuck: that is, if a running agent has some ongoing activity or an artifact has some operation to perform, then there is some rule that can be applied, hence execution will not lead to run-time errors (cf. Section 4). That is the standard *type soundness* property of statically typed languages.

An agent table GT and an artifact table AT map agent and artifact names to agent and class definitions. Following FJ [34], in presenting the type system and the operational semantics we assume fixed, global tables GT and AT. A *program* is a pair (GT, AT). We assume that these tables are

```
artifact C {
   obsprop int c;
   operation inc() :guard tr { .c = .c+1; }
}
agent U {
   Sns s;
   activity main(Counter c) :agenda ( usC(c) :pre tr  :pers tr ) { }
   activity usC(Counter c) :agenda() { use c.inc() :sns .s; }
}
```

Figure 3: Simplified versions of the artifact `Counter` and the agent `User` (cf. Figure 1)

well-formed, i.e., they contain precisely one entry for each agent/artifact mentioned in the program.

While presenting the type system, we will use the simplified versions of the artifact `Counter` and the agent `User` given in Figure 3 (cf. Figure 1) to illustrate some of the typing rules.

The typing rules for expressions, activity and agent declarations, operation/step and artifact declarations are given in Figure 4, 5 and 6, respectively. A type environment, Γ, is a finite mapping from variables to types, written $[\overline{\mathtt{x}} : \overline{\mathtt{T}}]$.

The typing judgements are as follows.

- $\boxed{\Gamma \vdash \mathtt{e} : \mathtt{T} \text{ IN } \mathtt{X}}$ meaning that, under the assumptions in Γ, the expression `e` has type `T` in the context of the artifact or agent `X`. The top of Figure 4 contains the rules for expressions that may occur in any context (artifact or agent). In rule [Tval], the function `typeOf` returns the type of a basic value (mapping `tr` and `fls` to `Bool`, and `unit` to `Unit`). The most interesting rules are: [TnewA], which ensures that when an artifact is created all its fields and properties are initialised with values of the appropriate type; and [TnewG], which ensures that when an agent is created the parameters of the activity `main` are initialised with values of the appropriate type.

  The middle of Figure 4 contains the rules for expressions that may occur inside artifacts only. The most interesting rules are: [Tnext], which ensures that when an operation step in invoked all the required parameters are supplied with the right type; and [Tsend], which ensures that when a perception is added to a sensor the type of the expression `e` is that of the values that may be associated with the label `l`, which is

Expressions that may occur in agent activities or artifact operations/steps:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \text{ IN } \mathtt{X} \quad \text{[Tvar]} \qquad \Gamma \vdash \mathtt{c} : \mathtt{typeOf(c)} \text{ IN } \mathtt{X} \quad \text{[Tval]}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \text{ IN } \mathtt{X} \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \text{ IN } \mathtt{X}}{\Gamma \vdash \mathtt{e}_1; \mathtt{e}_2 : \mathtt{T}_2 \text{ IN } \mathtt{X}}$$

$$\frac{\overline{\mathtt{U}}\,\overline{\mathtt{f}} \in \mathtt{A} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \text{ IN } \mathtt{X} \quad \overline{\mathtt{U}'}\,\overline{\mathtt{p}} \in \mathtt{A} \quad \Gamma \vdash \overline{\mathtt{e}'} : \overline{\mathtt{U}'} \text{ IN } \mathtt{X}}{\Gamma \vdash \mathtt{make\ A}(\overline{\mathtt{e}}, \overline{\mathtt{e}'}) : \mathtt{A} \text{ IN } \mathtt{X}} \text{ [TnewA]}$$

$$\frac{\mathtt{activity\ main}\ (\overline{\mathtt{T}}\,\overline{\mathtt{x}}) \cdots \in \mathtt{G} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \text{ IN } \mathtt{X}}{\Gamma \vdash \mathtt{spawn\ G}(\overline{\mathtt{e}}) : \mathtt{G} \text{ IN } \mathtt{X}} \text{ [TnewG]}$$

Expressions that may occur in artifact operations/steps only:

$$\frac{\overline{\mathtt{U}}\,\overline{\mathtt{f}} \in \mathtt{A} \quad \mathtt{f}_i \in \overline{\mathtt{f}}}{\Gamma \vdash .\mathtt{f}_i : \mathtt{U}_i \text{ IN } \mathtt{A}} \text{ [TfieldR]} \qquad \frac{\overline{\mathtt{U}}\,\overline{\mathtt{f}} \in \mathtt{A} \quad \mathtt{f}_i \in \overline{\mathtt{f}} \quad \Gamma \vdash \mathtt{e} : \mathtt{U}_i \text{ IN } \mathtt{A}}{\Gamma \vdash .\mathtt{f}_i = \mathtt{e} : \mathtt{U}_i \text{ IN } \mathtt{A}} \text{ [TfieldW]}$$

$$\frac{\overline{\mathtt{U}}\,\overline{\mathtt{p}} \in \mathtt{A} \quad \mathtt{p}_i \in \overline{\mathtt{p}}}{\Gamma \vdash .\mathtt{p}_i : \mathtt{U}_i \text{ IN } \mathtt{A}} \text{ [TprR]} \qquad \frac{\overline{\mathtt{U}}\,\overline{\mathtt{p}} \in \mathtt{A} \quad \mathtt{p}_i \in \overline{\mathtt{p}} \quad \Gamma \vdash \mathtt{e} : \mathtt{U}_i \text{ IN } \mathtt{A}}{\Gamma \vdash .\mathtt{p}_i = \mathtt{e} : \mathtt{T}_i \text{ IN } \mathtt{A}} \text{ [TprW]}$$

$$\frac{\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \text{ IN } \mathtt{A} \quad \mathtt{step\ o}(\overline{\mathtt{U}}\,\overline{\mathtt{x}}) \cdots \in \mathtt{A}}{\Gamma \vdash \mathtt{next\ o}(\overline{\mathtt{e}}) : \mathtt{A} \text{ IN } \mathtt{A}} \text{ [Tnext]} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{U} \text{ IN } \mathtt{A} \quad \mathtt{typeOfLab(l)} = \mathtt{U}}{\Gamma \vdash \mathtt{signal(l(e))} : \mathtt{U} \text{ IN } \mathtt{A}} \text{ [Tsend]}$$

Expressions that may occur in agent activities only:

$$\frac{\mathtt{activity\ a} \cdots \in \mathtt{G}}{\begin{array}{l}\Gamma \vdash \mathtt{started(a)} : \mathtt{Bool} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{completed(a)} : \mathtt{Bool} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{failed(a)} : \mathtt{Bool} \text{ IN } \mathtt{G}\end{array}} \text{ [TstateAct]} \qquad \Gamma \vdash \mathtt{fail} : \mathtt{T} \text{ IN } \mathtt{G} \quad \text{[Tfail]}$$

$$\frac{\mathtt{Sns}\ \overline{\mathtt{s}} \in \mathtt{G} \quad \mathtt{s} \in \overline{\mathtt{s}}}{\Gamma \vdash .\mathtt{s} : \mathtt{Sns} \text{ IN } \mathtt{G}} \text{ [Tsns]} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{Sns} \text{ IN } \mathtt{G} \quad \mathtt{typeOfLab(l)} = \mathtt{U}}{\Gamma \vdash \mathtt{sense\ e\ :filter\ l} : \mathtt{U} \text{ IN } \mathtt{G}} \text{ [Tperc]}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e}' : \mathtt{A} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{e} : \mathtt{Sns} \text{ IN } \mathtt{G} \\ \mathtt{operation\ o}(\overline{\mathtt{U}}\,\overline{\mathtt{x}}) \cdots \in \mathtt{A} \\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \text{ IN } \mathtt{G}\end{array}}{\Gamma \vdash \mathtt{use\ e'.o}(\overline{\mathtt{e}})\ \mathtt{:sns\ e} : \mathtt{Sns} \text{ IN } \mathtt{G}} \text{ [Top]} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \text{ IN } \mathtt{G} \quad \mathtt{typeOfLab(l)} = \mathtt{T}}{\begin{array}{l}\Gamma \vdash \mathtt{?memo(l)} : \mathtt{T} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{-memo(l)} : \mathtt{T} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{+memo(l(e))} : \mathtt{T} \text{ IN } \mathtt{G} \\ \Gamma \vdash \mathtt{memo(l)} : \mathtt{Bool} \text{ IN } \mathtt{G}\end{array}} \text{ [Tmm]}$$

$$\frac{\Gamma \vdash \mathtt{e}' : \mathtt{Sns} \text{ IN } \mathtt{G} \quad \Gamma \vdash \mathtt{e} : \mathtt{A} \text{ IN } \mathtt{G}}{\Gamma \vdash \mathtt{focus(e, e')} : \mathtt{Sns} \text{ IN } \mathtt{G}} \text{ [Tfocus]} \qquad \frac{\Gamma \vdash \mathtt{e}' : \mathtt{Sns} \text{ IN } \mathtt{G} \quad \Gamma \vdash \mathtt{e} : \mathtt{A} \text{ IN } \mathtt{G}}{\Gamma \vdash \mathtt{unfocus(e, e')} : \mathtt{Sns} \text{ IN } \mathtt{G}} \text{ [Tunfocus]}$$

$$\frac{\overline{\mathtt{U}}\,\overline{\mathtt{p}} \in \mathtt{A} \quad \Gamma \vdash \mathtt{e} : \mathtt{A} \text{ IN } \mathtt{G}}{\Gamma \vdash \mathtt{observe\ e.p}_i : \mathtt{U}_i \text{ IN } \mathtt{G}} \text{ [TpropA]}$$

Figure 4: Typing rules for expressions

$$\frac{\begin{array}{l}[\overline{\mathtt{x}} : \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{T} \text{ IN } \mathtt{G} \quad \mathtt{started(a)}, \mathtt{failed(a)}, \mathtt{completed(a)} \text{ are not in } \mathtt{e} \\ \text{(for all } i \in 1..n) \\ \quad \mathtt{SubAct}_i = \mathtt{a}_i(\overline{\mathtt{e}}^i) \text{ :pers } \mathtt{e}'_i \text{ :pre } \mathtt{e}''_i \quad \overline{\mathtt{e}}^i, \mathtt{e}'_i, \mathtt{e}''_i \text{ side effect free} \\ \quad \mathtt{activity\ a}_i\ (\overline{\mathtt{T}}^i\ \overline{\mathtt{x}}^i) \cdots \in \mathtt{G} \\ \quad [\overline{\mathtt{x}} : \overline{\mathtt{T}}] \vdash \overline{\mathtt{e}}^i : \overline{\mathtt{T}}^i \text{ IN } \mathtt{G} \quad [\overline{\mathtt{x}} : \overline{\mathtt{T}}] \vdash \mathtt{e}'_i : \mathtt{Bool} \text{ IN } \mathtt{G} \quad [\overline{\mathtt{x}} : \overline{\mathtt{T}}] \vdash \mathtt{e}''_i : \mathtt{Bool} \text{ IN } \mathtt{G}\end{array}}{\vdash \mathtt{activity\ a}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \text{:agenda}\ (\mathtt{SubAct}_1 \cdots \mathtt{SubAct}_n)\{\mathtt{e};\} \text{ OK IN } \mathtt{G}}$$

$$\frac{\begin{array}{l}[\overline{\mathtt{x}} : \overline{\mathtt{U}}] \vdash \mathtt{e}' : \mathtt{T}' \text{ IN } \mathtt{A} \\ [\overline{\mathtt{x}} : \overline{\mathtt{U}}] \vdash \mathtt{e} : \mathtt{Bool} \text{ IN } \mathtt{A} \quad \mathtt{e} \text{ side effect free}\end{array}}{\begin{array}{l}\vdash \mathtt{operation\ o}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \text{:guard e } \{\mathtt{e}';\} \text{ OK IN } \mathtt{A} \\ \vdash \mathtt{step\ o}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \text{:guard e } \{\mathtt{e}';\} \text{ OK IN } \mathtt{A}\end{array}}$$

Figure 5: Typing rules for activities and operations/steps agents

$$\frac{\vdash \overline{\mathtt{Act}} \text{ OK IN } \mathtt{G} \quad \mathtt{activity\ main}\ (\overline{\mathtt{T}}\,\overline{\mathtt{x}}) : \mathtt{agenda}(\overline{\mathtt{SubAct}})\{\mathtt{e};\} \in \overline{\mathtt{Act}}}{\vdash \mathtt{agent\ G}\ \{\ \mathtt{Sns}\ \overline{\mathtt{s}};\ \overline{\mathtt{Act}}\}\ \text{OK}}$$

$$\frac{\vdash \overline{\mathtt{Op}} \text{ OK IN } \mathtt{A} \quad \vdash \overline{\mathtt{Step}} \text{ OK IN } \mathtt{A}}{\vdash \mathtt{artifact\ A}\{\overline{\mathtt{U}}\ \overline{\mathtt{f}};\ \overline{\mathtt{V}}\ \overline{\mathtt{p}};\ \overline{\mathtt{Op}}\ \overline{\mathtt{Step}}\}\ \text{OK}}$$

Figure 6: Typing rules for agents and artifacts

specified by the function `typeOfLab`. Consider, for instance, operation `inc` of artifact `C` (Figure 3). Since the operation has no parameters, its body (`.c = .c+1`) can be typed under the empty set of assumptions. Assume that there are integer constants of type `int`, and the obvious typing rule for the sum of two integer expressions, say [Tsum]. The typing is as follows:

$$\dfrac{\mathtt{int\ c\ } \in \mathtt{C} \qquad \dfrac{\dfrac{}{\vdash 1 : \mathtt{int} \text{ IN } \mathtt{C}}\,[\text{Tval}] \quad \dfrac{\mathtt{int\ c\ } \in \mathtt{C}}{\vdash .\mathtt{c} : \mathtt{int} \text{ IN } \mathtt{C}}\,[\text{TprR}]}{\vdash .\mathtt{c} + 1 : \mathtt{int} \text{ IN } \mathtt{C}}\,[\text{Tsum}]}{\vdash .\mathtt{c} = .\mathtt{c} + 1 : \mathtt{int} \text{ IN } \mathtt{C}}\,[\text{TprW}]$$

The bottom of Figure 4 contains the rules for expressions that may occur inside agents only. The most interesting rules are: [Tfail], which states that `fail` may have any type; [Tperc], which states that if an agent senses a perception via one of its sensor then the type of the

obtained value is the one specified by the label `l` used as filter; and [Top], which ensures that when an agent uses an artifact the supplied sensor belongs to the agent, the invokes operations is supported by the artifact and all the parameters required by the operations are supplied with the right type. Consider, for instance, activity `usC` of agent `U` (Figure 3). Its body (`use c.inc() :sns .s`) can be typed under the type environment $\Gamma = \{\texttt{c:C}\}$, as follows:

$$\dfrac{\Gamma \vdash \texttt{c} : \texttt{U} \text{ IN } \texttt{U} \quad [\text{Tvar}] \qquad \dfrac{\texttt{Sns .s} \in \texttt{U}}{\Gamma \vdash \texttt{.s} : \texttt{Sns IN U}}\,[\text{Tsns}] \qquad \texttt{operation inc}() \cdots \in \texttt{U}}{\Gamma \vdash \texttt{use c.inc() :sns .s} : \texttt{Sns IN U}}\,[\text{Top}]$$

For an activity to be well typed, as shown in Figure 5, the expression (i.e. the body), must be well typed, and should not contain the predicates `started(a)`, `failed(a)`, and `completed(a)` that are meant to be just in the preconditions and persistence predicates. These predicates are used to coordinate the execution of sub-activities. Persistence and precondition predicates should be without side effects, that is, expressions which are (a boolean combination of) either a basic value `c`, or a parameter activity `x`, or one of the predicates `memo(l)`, `completed(a)`, `failed(a)`, and `started(a)`. The parameters of the activity, $\overline{\texttt{x}}$, may be used in the body of the activity and also in its sub-activities, providing a common state for the sub-activities. For instance, the formal parameter `c` of activity `usC` of agent `U` (Figure 3) is used in the body of the activity. The activity `usC` is well typed according to the rule at the top of Figure 5 (the typing of the body of activity `usC` has been showed before). The same rule can be used to establish that the activity `main` is well typed (since the body of the activity is empty, the first premise can be ignored).

Type checking of operations, Figure 5, checks that the body of the operation is well typed, and that its guard is of type boolean, and is side effect free. For an operation this means that the expression is (a boolean combination of) either a basic value `c`, or a parameter activity `x`, or a field access `.f`, or a property access `.p`. For instance, the operation `inc` of artifact `C` (Figure 3) is well typed according to the rule at the bottom of Figure 5 (the typing of the body of operation `inc` is shown before).

- $\boxed{\vdash \ldots \text{ OK}}$ meaning that the agent/artifact "..." is well typed. The

associated typing rules are in Figure 6, and just say that all the defined activities/operations/steps are well defined. Note that agents must contain the activity `main`. For instance, the rules can be used to establish that both the artifact `C` and the agent `U` (Figure 3) are well typed.

Definition 1 (Well-typed programs) *We write* $\vdash (\mathtt{GT}, \mathtt{AT})$ OK*, to be read "the program* $(\mathtt{GT}, \mathtt{AT})$ *is well-typed", to mean that the agents in* `GT` *and the artifacts in* `AT` *are well-typed.*

In the following we always assume that programs are well-typed.

3.3 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 of which has a unique identity, provided by a *reference*. The metavariable γ ranges over references to instance of agents, α over artifacts, σ over sensors.

The order in which the run-time expressions are evaluated inside agent/artifact instances is specified by using the standard technique of evaluation contexts [62].

3.3.1 Run-Time Expressions and Values

The *run-time expressions*, the expressions used during evaluation, do not contain variables, as variables are dynamically replaced by references, as we will see when defining the operational semantics. However, run-time expressions may contain references to agents, artifacts, or sensors. Therefore run-time expressions are defined by replacing, in the productions of the grammar for expression in Figure 2, `x` with ι.

The run-time *values*, ranged over by `v` and `w`, are defined as follows

$$\mathtt{v} \ ::= \ \iota \ \mid \ \mathtt{c}$$

That is, a value is either a reference to an agent/artifact/sensor instance or a basic value.

In the rest of the paper we use "expression" to refer to run-time expressions, whereas we use "user-level expression" to mean that the expression is generated by the grammar in Figure 2.

3.3.2 Configurations

Configurations are non-empty sets of sensor, agent, or artifact instances.
Sensor instances are represented by

$$\sigma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}} \rangle^{\mathtt{Sns}}$$

where σ is the instance identifier, and $\overline{\mathtt{l}}\,\overline{\mathtt{v}}$ is the queue of label/value associations representing the events generated (and not yet perceived) on the sensor.
Agent instances are represented by

$$\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R} \rangle^{\mathtt{G}}$$

where γ is the agent identifier, $\mathtt{G}$ is the type of the agent, $\overline{\mathtt{l}}\,\overline{\mathtt{v}}$ is the content of the *memo-space*, $\overline{\sigma}$ is the sequence of references to the instances of the sensors that the agent uses perceive, and $\mathtt{R}$ is the state of the activity `main` that was started when the agent was created. All the activities that the agent will be involved in are subactivities (possibly not direct) of the activity `main`, as we can see from the example in Section 2. The sensor instances in $\overline{\sigma}$ are in one-to-one correspondence with the sensor variables declared in the agent (see the rule at the bottom of Figure 5), and are needed since every agent uses its own set of sensor instances.
An *instance of an activity*, $\mathtt{R}$, 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.

$$\mathtt{R} \;\; ::= \;\; \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_n]\{\mathtt{e}\} \;\; | \;\; \mathtt{failed}^{\mathtt{a}}$$

The name of the activity is $\mathtt{a}$, $\overline{\mathtt{v}}$ are the actual parameters of the current activity instance, $\mathtt{Sr}_1 \cdots \mathtt{Sr}_n$ is the set of sub-activities running, and $\mathtt{e}$ 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 $\mathtt{failed}^{\mathtt{a}}$ we say that activity $\mathtt{a}$ *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 $n = 0$ the evaluation of the body $\mathtt{e}$ starts.

The state of a running sub-activity is represented by:

$$\mathtt{Sr} \;\; ::= \;\; \mathtt{a}(\overline{\mathtt{e}})\langle \mathtt{e}_1, \mathtt{e}_2 \rangle \;\; | \;\; \mathtt{R}$$

During the evaluation of the precondition and the persistence predicate a sub-activity is represented by the term, $\mathtt{a}(\overline{\mathtt{v}})\langle \mathtt{e}_1, \mathtt{e}_2\rangle$ where $\mathtt{e}_1$ (resp., $\mathtt{e}_2$) represents the state of evaluation of the persistence (resp., precondition) predicate. The persistence predicate is evaluated first and if it evaluates to `tr`, then the evaluation of $\mathtt{e}_2$ is started.

In the operational semantics, we will see that when an activity starts for the first time, the persistence predicate of all its sub-activities is set to `tr` (rule [SCH]), so that sub-activities are executed at least once (in accordance with the truth of their precondition). A subsequent scheduling will depend on the truth of the persistence predicate. In case, the evaluation of the persistence predicate $\mathtt{e}_1$ is `fls` the sub-activity is removed from the set of running sub-activities.

The precondition $\mathtt{e}_2$ is evaluated only in case the persistence predicate $\mathtt{e}_1 = \mathtt{tr}$. If $\mathtt{e}_2$ evaluates to `tr`, the term $\mathtt{a}(\overline{\mathtt{v}})\langle \mathtt{tr}, \mathtt{tr}\rangle$ is replaced with the initial state of the evaluation of the activity `a` with parameters $\overline{\mathtt{v}}$, which is represented by `R`. Instead, if $\mathtt{e}_2$ evaluates to `fls` the evaluation of the precondition of `a` is rescheduled.

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 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 $\mathtt{O}_i$, $1 \leq i \leq n$, 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 $\mathtt{O}_1$. Artifacts are single threaded and (differently from agents that may have more than one activity running at the same time) only the operation $\mathtt{O}_n$ is in execution at any time.

A *running operation*, `O`, is defined as follows.

$$\mathtt{O} \;::=\; (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathtt{e}_1\rangle\{\mathtt{e}_2\}) \;\mid\; (\sigma, \mathtt{o}[\mathtt{St}_1 \cdots \mathtt{St}_q]) \;\; p \geq 0,\, q \geq 1$$

The first kind of running operation is an operation that is *evaluating its guard or its body*, whereas the second kind specifies an operation that is *evaluating the guards of its steps* (after having evaluated its body). For a running operation `O`, 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 send events generated during the execution of the operation by `signal`, $[\mathtt{St}_1 \cdots \mathtt{St}_p]$ is the set of *steps* generated by the evaluation of the

body $\mathtt{e}_2$ of the operation. Moreover, $\overline{\mathtt{v}}$ are the actual parameters on which the operation was started.
For the running operation $(\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathtt{e}_1 \rangle \{\mathtt{e}_2\})$, if $\mathtt{e}_1$ is different from `tr` or `fls` the operation is evaluating its guard $\mathtt{e}_1$. If $\mathtt{e}_1 = \mathtt{fls}$ then the operation is dequeued and then enqueued, so that when it will be rescheduled it will restart evaluating its guard. If $\mathtt{e}_1 = \mathtt{tr}$ then the operation is either evaluating its body $\mathtt{e}_2$ or when $\mathtt{e}_2$ is a value: if $p = 0$ (no steps were generated), then the operation is successfully completed and therefore dequeued, otherwise, i.e., $(\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathtt{tr} \rangle \{\mathtt{v}\})$ where $p \geq 1$, the operation is dequeued and $(\sigma, \mathtt{o}[\mathtt{St}_1 \cdots \mathtt{St}_p])$ is enqueued, so that when it will be scheduled the evaluation of the guards of its steps will start.
Operation steps

$$\mathtt{St} \ ::= \ \mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{e} \rangle$$

in addition to the guard `e`, specify the actual parameters of the operation step.

3.3.3 Evaluation Contexts and Redexes

Evaluations contexts are used for specifying the evaluation order of various constructs. In particular, we will use it for expressions, and in this case, as Proposition 1 states, they are deterministic. They identify the first redex to be reduced. We will also use evaluation contexts later for activities and operations. For operations these contexts are again deterministic, whereas for activities they are not, since if there are many sub-activities, the choice of which one to evaluate first is not fixed. However, once a sub-activity is chosen, then the first expression to evaluate is uniquely determined.

An *evaluation context for expressions* $\mathcal{E}$ is an expression with one hole $[\![\ ]\!]$ in it. The term $\mathcal{E}[\![\mathtt{e}]\!]$ denotes the expression obtained from the context $\mathcal{E}$ by substituting its hole with the expression `e`. Let $\varphi \in \{\mathtt{focus}, \mathtt{unfocus}\}$. Evaluation contexts for expressions are defined as follows

$$\begin{array}{rcl}
\mathcal{E} & ::= & [\![\ ]\!] \ \mid \ \mathtt{spawn}\,\mathtt{G}(\overline{\mathtt{v}}\,\mathcal{E}\,\overline{\mathtt{e}}) \ \mid \ \mathtt{make}\,\mathtt{A}(\overline{\mathtt{v}}\,\mathcal{E}\,\overline{\mathtt{e}}) \ \mid \ \mathcal{E};\mathtt{e} \\
& \mid & .\mathtt{f} = \mathcal{E} \ \mid \ \mathtt{next}\ \mathtt{o}(\overline{\mathtt{v}}\,\mathcal{E}\,\overline{\mathtt{e}}) \ \mid \ \mathtt{signal}(\mathtt{l}(\mathcal{E})) \ \mid \ .\mathtt{p} = \mathcal{E} \\
& \mid & \mathtt{use}\ \mathtt{e}.\mathtt{o}(\overline{\mathtt{e}})\ \mathtt{:sns}\ \mathcal{E} \ \mid \ \mathtt{use}\ \mathcal{E}.\mathtt{o}(\overline{\mathtt{e}})\ \mathtt{:sns}\ \mathtt{v} \\
& \mid & \mathtt{use}\ \mathtt{v}.\mathtt{o}(\overline{\mathtt{v}}\,\mathcal{E}\,\overline{\mathtt{e}})\ \mathtt{:sns}\ \mathtt{v} \\
& \mid & \mathtt{+memo}(\mathtt{l}(\mathcal{E})) \ \mid \ \mathtt{observe}\ \mathcal{E}.\mathtt{p} \\
& \mid & \varphi(\mathcal{E}, \mathtt{e}) \ \mid \ \varphi(\mathtt{v}, \mathcal{E}) \ \mid \ \mathtt{sense}\ \mathcal{E}\ \mathtt{:filter}\ \mathtt{l}
\end{array}$$

Evaluation contexts for expressions specify the standard call-by-value evaluation, where in general parameters are evaluated left-to-right. *Redexes*, ranged over by `rdx`, are the elements of the set $\mathcal{X} = \mathcal{X}_{\mathtt{A}} \cup \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$, where:

$$\begin{array}{lcl}
\mathcal{X}_{\mathtt{X}} & = & \{\mathtt{spawn\ G}(\overline{\mathtt{v}}), \mathtt{make\ A}(\overline{\mathtt{v}}), \mathtt{v}; \mathtt{e}\} \\
\mathcal{X}_{\mathtt{A}} & = & \{\mathtt{.f}, \mathtt{.f} = \mathtt{v}, \mathtt{.p}, \mathtt{.p} = \mathtt{v}, \mathtt{next\ o}(\overline{\mathtt{v}}), \mathtt{signal}(\mathtt{l}(\mathtt{v}))\} \\
\mathcal{X}_{\mathtt{G}} & = & \{\mathtt{.s}, \mathtt{use\ v.o}(\overline{\mathtt{v}})\ \mathtt{:sns\ v}, \mathtt{sense\ v\ :filter\ l}, \\
 & & \mathtt{?memo}(\mathtt{l}), \mathtt{-memo}(\mathtt{l}), \mathtt{+memo}(\mathtt{l}(\mathtt{v})), \mathtt{memo}(\mathtt{l}), \\
 & & \mathtt{focus}(\mathtt{v}, \mathtt{v}), \mathtt{unfocus}(\mathtt{v}, \mathtt{v}), \mathtt{observe\ v.p}, \\
 & & \mathtt{completed}(\mathtt{a}), \mathtt{failed}(\mathtt{a}), \mathtt{fail}\}
\end{array}$$

The set $\mathcal{X}_{\mathtt{X}}$ contains the redexes that may occur in any context (operations/steps of artifact instances, or run-time activities of agent instances), the set $\mathcal{X}_{\mathtt{A}}$ contain the redexes that may occur only in (operations/steps of) artifact instances, and the set $\mathcal{X}_{\mathtt{G}}$ contains the redexes that may occur only in (activities of) agent instances.

The following proposition states that evaluation contexts and redexes decompose expressions in a unique way (the proof is straightforward by structural induction on expressions). This proposition assumes that our set of definitions is well typed, which is the assumption we made at the end of the previous section.

Proposition 1 (Unique decomposition of expressions) *Given an expression* `e`*, either* `e` *is a value or there is a unique evaluation context* $\mathcal{E}$ *such that* $\mathtt{e} = \mathcal{E}[\![\mathtt{rdx}]\!]$ *for some* `rdx`*.*

Evaluation contexts for (expressions in) activities are defined as follows

$$\begin{array}{lcl}
\mathcal{R} & ::= & \mathtt{a}(\overline{\mathtt{w}})[\,]\{\mathcal{E}\} \ \mid\ \mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathcal{R}\ \ \overline{\mathtt{Sr}}]\{\mathtt{e}\} \\
 & \mid & \mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}\ \ \overline{\mathtt{Sr}}]\{\mathtt{e}\} \\
\mathcal{P} & ::= & \mathtt{a}(\overline{\mathtt{v}})\langle\mathcal{E}, \mathtt{e}\rangle \ \mid\ \mathtt{a}(\overline{\mathtt{v}})\langle\mathtt{tr}, \mathcal{E}\rangle \ \mid\ \mathtt{a}(\overline{\mathtt{v}}\,\mathcal{E}\,\overline{\mathtt{e}})\langle\mathtt{e}, \mathtt{e}'\rangle
\end{array}$$

If an activity does not have sub-activities that have to be evaluated, then the context selects the subexpression of its body that needs to be evaluated, otherwise it (non-deterministically) selects one of its sub-activities with the context $\mathcal{P}$. The sub-activity may be a running activity (in which case one of its subexpressions is evaluated), or one which specifies that the evaluation of its parameters or persistence predicate or precondition is not yet completed. In this case: first the parameters are evaluated left to right, then the persistence predicate, and finally the precondition.

For activities we also define a context $\mathcal{S}$ that (non-deterministically) selects a sub-activity.

$$\mathcal{S} \;::=\; (\![\ ]\!) \;\mid\; \mathtt{a}(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \mathcal{S}\ \overline{\mathtt{Sr}}]\{\mathtt{e}\}$$

This context is needed when we want to identify a whole sub-activity, such as when we have to schedule execution of sub-activities, whereas the context $\mathcal{R}$ selects an expression within an activity.

Operation evaluation contexts are defined as follows

$$\begin{array}{rcl} \mathcal{O} & ::= & (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle\mathcal{E}\rangle\{\mathtt{e}\}) \;\mid\; (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle\mathtt{tr}\rangle\{\mathcal{E}\}) \\ & \mid & (\sigma, \mathtt{o}[\overline{\mathtt{o}(\overline{\mathtt{v}})\langle\mathtt{fls}\rangle}\ \ \mathtt{o}'(\overline{\mathtt{v}})\langle\mathcal{E}\rangle\ \ \overline{\mathtt{St}}]) \end{array}$$

where with $\overline{\mathtt{o}(\overline{\mathtt{v}})\langle\mathtt{fls}\rangle}$ we denote a sequence of steps all with guard equals to `fls`. For an operation we first evaluate the guard expression, and in case this is `tr` we also evaluate its body. Guards of steps are evaluated left to right while the guard evaluates to false.

Finally, to account for the redexes that may occur in activities and operation bodies ($\mathtt{spawn}\,\mathtt{G}(\overline{\mathtt{v}})$, $\mathtt{make}\,\mathtt{A}(\overline{\mathtt{v}})$, and $\mathtt{v};\mathtt{e}$) we define *agent/artifact instance evaluation contexts*, by

$$\mathcal{K} \;::=\; \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathcal{R}\rangle^{\mathtt{G}} \;\mid\; \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma},\ \overline{\mathtt{O}}\ \mathcal{O}\rangle^{\mathtt{A}}$$

Note that, for artifact instances, the context selects the last operation of the sequence of running operations, which is the running one.

3.3.4 Initial Configurations and Reduction Relation

Evaluation of an *executable program* starts from its *initial configuration* defined as follows.

Definition 2 (Initial configuration) *A program* $(\mathtt{GT}, \mathtt{AT})$ *is* executable *if the agent table* `GT` *contains an entry for an agent with distinguished name* `Main` *of the shape*

$$\mathtt{agent}\ \mathtt{Main}\ \{\ \mathtt{activity}\ \mathtt{main}\,()\ \ \mathtt{:agenda}\,()\ \{\mathtt{e};\}\ \}$$

The initial configuration associated to the executable program $(\mathtt{GT}, \mathtt{AT})$ *is*

$$\gamma = \langle \emptyset, \emptyset, \mathtt{main}(\ )[\ ]\{\mathtt{e}\}\rangle^{\mathtt{Main}}$$

for some reference γ.

$$\frac{\mathtt{M} \equiv \mathtt{M}_0 \qquad \mathtt{M}_0 \longrightarrow \mathtt{M}_1}{\mathtt{M} \Rightarrow \mathtt{M}_1} \quad \text{[EMPTYCONT]}$$

$$\frac{\mathtt{M} \equiv \mathtt{M}' \mid \mathtt{M}_0 \qquad \mathtt{M}_0 \longrightarrow \mathtt{M}_1}{\mathtt{M} \Rightarrow \mathtt{M}' \mid \mathtt{M}_1} \quad \text{[CONT]}$$

Figure 7: Reduction rules: configuration congruence

For example, for our system of Figure 1, the initial configuration is:

$$\gamma_M = \langle \emptyset, \emptyset, \texttt{main}(\,)[\,]\{\texttt{spawn Init}(\texttt{make Counter}(0));\}\rangle^{\texttt{Main}}$$

Note that, we can write the initial configuration as:

$$\mathcal{K}[\![\texttt{make Counter(0)}]\!] \tag{1}$$

where $\mathcal{K}$ is

$$\gamma_M = \langle \emptyset, \emptyset, \texttt{main}(\,)[\,]\{\texttt{spawn Init}([\![\,]\!])\}\rangle^{\texttt{Main}}$$

In the following, to shorten configurations, we will use for names of agents and artifacts just the first letter of their name.

The reduction relation has the form $\mathtt{M} \Rightarrow \mathtt{M}'$ meaning that the configuration $\mathtt{M}$ reduces to configuration $\mathtt{M}'$ in one step, where configurations are defined as follows:

$$\begin{array}{lcl} \mathtt{M} & ::= & \mathtt{I} \mid (\mathtt{M} \mid \mathtt{M}) \\ \mathtt{I} & ::= & \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}} \mid \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_n\rangle^{\mathtt{A}} \mid \sigma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}\rangle^{\mathtt{Sns}} \end{array}$$

and the *configuration congruence relation*, $\equiv$, formalizes the fact that configurations represents non-empty sets of instances, that is:

$$(\mathtt{M} \mid \mathtt{M}') \equiv (\mathtt{M}' \mid \mathtt{M}) \qquad ((\mathtt{M} \mid \mathtt{M}') \mid \mathtt{M}'') \equiv (\mathtt{M} \mid (\mathtt{M}' \mid \mathtt{M}''))$$

We use the congruence relation between configurations in the definition of the relation $\Rightarrow$, see rules [EMPTYCONT] and [CONT] of Figure 7. Such rules rearrange agents/artifacts/sensors instances in order to apply the reduction $\longrightarrow$ for *minimal configurations*. Minimal configurations are configurations containing exactly the agent/artifact/sensor instances involved in the reduction. We write $\Rightarrow^\star$ for the reflexive and transitive closure of $\Rightarrow$.

The rules in Figure 8 generate agents and artifacts. Creation may occur in any context. Creation of an agent, rule [AGN], in addition to creating a binding between a fresh identifier γ and an instance of an agent, creates m

instances of sensors corresponding to the sensor identifiers declared in the agent. The `main` activity is also started, setting up its sub-activities. Since sub-activities have to be evaluated at least once, the persistence predicate is set to `tr`. Parameters, persistence predicate and preconditions of sub-activities may contain the formal parameters of the activity, therefore they are substituted with the actual parameters of `spawn G`. For an artifact, rule [ART], we create a binding between a fresh artifact identifier and an artifact instance. Moreover, the fields and properties of the artifact are initialized to the actual parameters of `make A`.

Take our initial configuration (1). The first (and only) reduction possible is using rule [ART], that is

$$(1) \longrightarrow \ \mathcal{K}[\![\alpha_{\mathtt{C}}]\!] \mid \alpha_{\mathtt{C}} = \langle \emptyset, \mathtt{count} = 0, \emptyset, \emptyset \rangle^{\mathtt{C}}$$

and

$$\mathcal{K}[\![\alpha_{\mathtt{C}}]\!] = \mathcal{K}'[\![\mathtt{spawn\ I}(\alpha_{\mathtt{C}})]\!] \quad \text{where } \mathcal{K}' \text{ is } \gamma_M = \langle \emptyset, \emptyset, \mathtt{main}(\,)[\,]\{[\![\,]\!]\} \rangle^{\mathtt{M}}$$

applying rule [AGN] to $\mathcal{K}'[\![\mathtt{spawn\ I}(\alpha_{\mathtt{C}})]\!]$ we get

$$\gamma_M = \langle \emptyset, \emptyset, \mathtt{main}(\,)[\,]\{\gamma_I\} \rangle^{\mathtt{M}} \mid \gamma_I = \langle \emptyset, \emptyset, \mathtt{main}(\alpha_{\mathtt{C}})[\,]\{\mathtt{e}_1; \mathtt{e}_2; \mathtt{e}_3\} \rangle^{\mathtt{I}} \tag{2}$$

where (i) $\mathtt{e}_1$ is $\mathtt{spawn\ O}(\alpha_{\mathtt{C}})$, (ii) $\mathtt{e}_2$ is $\mathtt{spawn\ U}(\alpha_{\mathtt{C}})$, and (iii) $\mathtt{e}_3$ is $\mathtt{spawn\ U}(\alpha_{\mathtt{C}})$.

Most of the time configurations have more than one instance of agent, artifact, or sensor in parallel. So to reduce configurations we apply [CONT] (or [EMPTYCONT] if we just need to rearrange the term) with the chosen rule applied to the minimal configuration on its premises. In the following we will just show examples of application of the rule to minimal configurations. The means by which $\Rightarrow$ is obtained should be obvious. Applying [AGN] to γ_I we generate a new agent instance of type `Observer`, and since there is a sensor declared in its definition, we also generate a sensor instance. The main activity of the agent `Observer` has two sub-activities: `prepare` and `monitoring` whose persistence predicate is initialized to `tr` and the precondition to the one specified in the definition of the sub-activity.

$$\begin{array}{l} \mathcal{K}''[\![\mathtt{spawn\ O}(\alpha_{\mathtt{C}})]\!] \longrightarrow \\ \mathcal{K}''[\![\gamma_O]\!] \mid \gamma_O = \langle \emptyset, \sigma, \mathtt{main}(\alpha_{\mathtt{C}})[\mathtt{S1\ S2}]\{\} \rangle^{\mathtt{O}} \mid \sigma = \langle \emptyset \rangle^{\mathtt{Sns}} \end{array} \tag{3}$$

where $\mathcal{K}''$ is

$$\gamma_O = \langle \emptyset, \emptyset, \mathtt{main}(\,)[\,]\{[\![\,]\!]; \mathtt{spawn\ U}(\alpha_{\mathtt{C}});\ \mathtt{spawn\ U}(\alpha_{\mathtt{C}})\} \rangle^{\mathtt{I}}$$

$$\frac{\begin{array}{c}\gamma \text{ fresh} \qquad \texttt{activity main}\ (\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{:agenda(SubAct}_1 \cdots \texttt{SubAct}_n)\{\texttt{e};\} \in \texttt{G} \\ (\text{for all } i \in 1..n) \quad \texttt{SubAct}_i = \texttt{a}_i(\overline{\texttt{e}}^i)\ \texttt{:pers}\ \texttt{e}'_i\ \texttt{:pre}\ \texttt{e}''_i \\ \texttt{Sr}_i = \texttt{a}_i(\overline{\texttt{e}}^i[\overline{\texttt{v}}/\overline{\texttt{x}}])\langle \texttt{tr}, \texttt{e}''_i[\overline{\texttt{v}}/\overline{\texttt{x}}]\rangle \\ \texttt{Sns}\ \texttt{s}_1, \ldots, \texttt{s}_m \in \texttt{G} \qquad \overline{\sigma} = \sigma_1 \cdots \sigma_m \qquad (\text{for all } j \in 1..m) \quad \sigma_j \text{ fresh}\end{array}}{\begin{array}{l}\mathcal{K}[\![\texttt{spawn G}(\overline{\texttt{v}})]\!] \longrightarrow \\ \quad \mathcal{K}[\![\gamma]\!] \mid \gamma = \langle \emptyset, \overline{\sigma}, \texttt{main}(\overline{\texttt{v}})[\texttt{Sr}_1 \cdots \texttt{Sr}_n]\{\texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}]\}\rangle^{\texttt{G}} \\ \quad\quad \mid \sigma_1 = \langle \emptyset \rangle^{\texttt{Sns}} \mid \cdots \mid \sigma_m = \langle \emptyset \rangle^{\texttt{Sns}}\end{array}} \quad [\text{AGN}]$$

$$\frac{\alpha \text{ fresh} \qquad \overline{\texttt{U}}\ \overline{\texttt{f}}; \overline{\texttt{V}}\ \overline{\texttt{p}} \in \texttt{A}}{\mathcal{K}[\![\texttt{make A}(\overline{\texttt{v}}\ \overline{\texttt{w}})]\!] \longrightarrow \mathcal{K}[\![\alpha]\!] \mid \alpha = \langle \overline{\texttt{f}} = \overline{\texttt{v}}, \overline{\texttt{p}} = \overline{\texttt{w}}, \emptyset, \emptyset \rangle^{\texttt{A}}} \quad [\text{ART}]$$

$$\mathcal{K}[\![\texttt{v}; \texttt{e}]\!] \longrightarrow \mathcal{K}[\![\texttt{e}]\!] \quad [\text{SEQ}]$$

Figure 8: Reduction rules: agent and artifact instances creation, sequential composition

and

- $\texttt{S1} = \texttt{prepare}(\alpha_{\texttt{C}})\langle \texttt{tr}, \texttt{tr} \rangle$, and
- $\texttt{S2} = \texttt{monitoring}(\alpha_{\texttt{C}})\langle \texttt{tr}, \texttt{completed}(\texttt{prepare}) \rangle$.

Note that the persistence predicate of the sub-activity `prepare` is initialized to `tr` (even though it was declared to be `fls`) so that the sub-activity is scheduled a first time. As we will see later, when at the end of its execution it is rescheduled again, then its persistence predicate will be initialized to its definition. This causes exactly one execution of the sub-activity `prepare`. Let us now assume that we apply [SEQ] that removes the value in the hole of the agent γ_O. At this point the evolution of the system can go on either spawning a `User` agent or starting the evaluation or scheduling the execution of the sub-activity `prepare`, see rule [SCH] of Figure 10. Let us assume that we spawn the two `User` agents and produce the following configuration:

$$\begin{array}{l}\gamma_M = \langle \emptyset, \emptyset, \texttt{main}(\ )[\ ]\{\ \}\rangle^{\texttt{M}} \mid \gamma_I = \langle \emptyset, \emptyset, \texttt{main}(\alpha_{\texttt{C}})[\ ]\{\ \}\rangle^{\texttt{I}} \mid \\ \alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \emptyset, \emptyset \rangle^{\texttt{C}} \mid \\ \gamma_O = \langle \emptyset, \sigma, \texttt{main}(\alpha_{\texttt{C}})[\texttt{S1 S2}]\{\}\rangle^{\texttt{O}} \mid \gamma_{U1} = \langle \emptyset, \sigma_1, \texttt{main}(\alpha_{\texttt{C}})[\texttt{SA}_{\texttt{U1}}]\{\}\rangle^{\texttt{U}} \mid \\ \gamma_{U2} = \langle \emptyset, \sigma_2, \texttt{main}(\alpha_{\texttt{C}})[\texttt{SA}_{\texttt{U2}}]\{\}\rangle^{\texttt{U}} \mid \sigma = \langle \emptyset \rangle^{\texttt{Sns}} \mid \sigma_1 = \langle \emptyset \rangle^{\texttt{Sns}} \mid \sigma_2 = \langle \emptyset \rangle^{\texttt{Sns}}\end{array} \quad (4)$$

where $\texttt{SA}_{\texttt{U1}}$ and $\texttt{SA}_{\texttt{U2}}$ denote $\texttt{usingCount}(\alpha_{\texttt{C}})\langle \texttt{tr}, \texttt{tr} \rangle$.

In Figure 9 we present the evaluation rules for expressions that may occur only inside activities (and therefore agents). The first three rules deal

with agent memos. Rule [MMmatch] is for reading, removing and querying memos in the agent memo-space when the memo is present. When the specified memo is not present in the agent memo-space, rule [MMmismatch] generates a failure in case we try to read or remove, and returns `fls` if we queried for presence of the memo via the predicate `memo(l)`. Rule [MMins] inserts the memo in the memo-space.

Rule [SNS] returns a reference to the instance of the sensor corresponding to the identifier `s` (see rule [AGN] for the initialization of the sensors). If an agent is well-typed there is always a sensor instance corresponding to `s`. Rule [PER] extracts the first value associated with a specific label from a sensor, in case no association for `l` is found the rule is not applicable, therefore the activity is blocked. In this case, the presence of the label is a run-time condition that may be used for synchronizing agent activities.

Rule [GETA] reads property $\mathtt{p}_i$ of agent α. Rule [FOC], and [UNFOC] start/stop focussing on the events of an artifact by inserting/removing the sensor in the list of sensors of the artifacts.

Rule [USE] starts the operation `o` on the artifact α with parameters $\overline{\mathtt{v}}$ defining σ as the operation sensor. The operation is enqueued in the queue of operations for the artifact α. Therefore, as we will see from the rules of Figure 12, it will be the last one to be scheduled for execution.

Rule [FAIL] propagates a failure in an activity by replacing the activity with a failed activity—this is a very primitive way of dealing with failure, simplifying the actual SIMPA management of failures which is based on standard `try/catch` constructs.

The last three rules check the state of the execution of sibling sub-activities. The predicate `completed(a)`, rule [COMPL], checks the presence of the sub-activity `a` in the list of sub-activities. (When an activity is started the evaluation of the sub-activities in its agenda is started by putting them in the list of sub-activities, and when the evaluation of a sub-activity is completed, the sub-activity is removed, so the list contains the sub-activities not yet completed.) The auxiliary function `actNames()` is defined as follows

$$\begin{aligned}\mathtt{actNames}(\overline{\mathtt{Sr}}) \;=\; \{\mathtt{a} \mid\; & \mathtt{a}(\cdots)[\cdots]\{\cdots\} \in \overline{\mathtt{Sr}} \\ & \text{or } \mathtt{failed}^{\mathtt{a}} \in \overline{\mathtt{Sr}} \\ & \text{or } \mathtt{a}(\cdots)\langle\cdots,\cdots\rangle \in \overline{\mathtt{Sr}}\}\end{aligned}$$

Rule [STARTED] checks whether `a` is in the list of sub-activities and it is a running activity. The auxiliary function `actStarted()` is defined as follows

$$\mathtt{actStarted}(\overline{\mathtt{Sr}}) = \{\mathtt{a} \mid \mathtt{a}(\cdots)[\cdots]\{\cdots\} \in \overline{\mathtt{Sr}}\}$$

Finally, rule [FAILED] checks whether or not the sub-activity failed.

In Figure 10 we present the rules for scheduling the execution of sub-activities. When an activity starts its execution, the sub-activities in its todo list are put in the pool of sub-activities that needs to be evaluated with `tr` as persistence predicate. Therefore, the evaluation of their precondition can start. If the evaluation of the precondition produces `tr`, rule [SCH] starts the evaluation of a sub-activity, named `a`, whose persistence predicate and precondition are true. As already mentioned, since the sub-activities of `a` have to be evaluated at least once, the persistence predicate is set to `tr`.

Preconditions may evaluate to `fls`, in this case, rule [PREC] restarts the evaluation of the precondition. The body of the precondition is found in the agenda of the activity `a`. Since this precondition may contain the formal parameters of the activity `a` of which $\mathtt{a}_i$ is a sub-activity, in the expression the formal parameters of `a` must be substituted with the actual parameters at the time he activity was started.

Rule [DISP] removes a sub-activity whose persistence predicate is false. Note that, from rule [SCH], the persistence predicate may be `fls` only after the first evaluation of the sub-activity. Indeed sub-activities that have to be executed only once have persistence predicate `fls`.

Rule [END-SA] restart the evaluation of the parameters, persistence predicate and precondition of a completed sub-activity, $\mathtt{a}'$. The value `v` resulting from the evaluation of the sub-activity is inessential. Note that in this case, since the sub-activity $\mathtt{a}'$ has already been evaluated at least once, the persistence predicate to be evaluated is the one specified in the agenda (with the substitution of parameters).

Going back to our configuration (4) we can schedule, applying [SCH], either the sub-activity `prepare` of the agent γ_O, or one of the sub-activities `usingCount`, of the agents γ_{U1} or γ_{U2} because all these sub-activities have both predicates equal to `tr`. Assume we schedule `prepare` of the agent γ_O. In the configuration of (4) the only think that changes is that sub-activity `S1` (`prepare`) from a sub-activity that was evaluating its predicates becomes a sub-activity that is executing its body

$$\gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\mathtt{main}(\alpha_\mathtt{C})[\mathtt{S1\ S2}]\{\ \}]\!])\rangle^{\mathtt{0}} \longrightarrow \gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\mathtt{main}(\alpha_\mathtt{C})[\mathtt{S1'\ S2}]\{\ \}]\!])\rangle^{\mathtt{0}} \quad (5)$$

where the evaluation context $\mathcal{S}$ is $([\![\ ]\!])$, and $\mathtt{S1'}$ is

$$\mathtt{prepare}(\alpha_\mathtt{C})[\ ]\{\mathtt{focus}(\alpha_\mathtt{C}, \mathtt{s});\}$$

$$\frac{\overline{\mathtt{l}} = \mathtt{l}_1 \cdots \mathtt{l}_n \qquad (\text{exists } i \in 1..n)\ (\text{for all } j \in i+1..n) \quad \mathtt{l} = \mathtt{l}_i \wedge \mathtt{l} \neq \mathtt{l}_j}{\begin{array}{c}\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{?memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{v}_i]\!]\rangle^{\mathtt{G}} \\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{tr}]\!]\rangle^{\mathtt{G}} \\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{-memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow \\ \qquad \gamma = \langle \mathtt{l}_1\,\mathtt{v}_1 \cdots \mathtt{l}_{i-1}\,\mathtt{v}_{i-1}\mathtt{l}_{i+1}\,\mathtt{v}_{i+1} \cdots \mathtt{l}_n\,\mathtt{v}_n, \_, \mathcal{R}[\![\mathtt{v}_i]\!]\rangle^{\mathtt{G}}\end{array}} \quad [\text{MMmatch}]$$

$$\frac{\overline{\mathtt{l}} = \mathtt{l}_1 \cdots \mathtt{l}_n \qquad (\text{for all } i \in 1..n) \quad \mathtt{l} \neq \mathtt{l}_i}{\begin{array}{c}\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{?memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{fail}]\!]\rangle^{\mathtt{G}} \\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{-memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{fail}]\!]\rangle^{\mathtt{G}} \\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{memo(l)}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{fls}]\!]\rangle^{\mathtt{G}}\end{array}} \quad [\text{MMmismatch}]$$

$$\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{+memo(l(v))}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \mathtt{l}\,\mathtt{v}\ \ \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \_, \mathcal{R}[\![\mathtt{v}]\!]\rangle^{\mathtt{G}} \quad [\text{MMins}]$$

$$\frac{\mathtt{Sns}\ \mathtt{s}_1, \ldots, \mathtt{s}_n \in \mathtt{G} \qquad \bar{\sigma} = \sigma_1 \cdots \sigma_n \qquad (\text{exists } i \in 1..n) \quad \mathtt{s} = \mathtt{s}_i}{\gamma = \langle \_, \overline{\sigma}, \mathcal{R}[\![\mathtt{.s}]\!]\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \_, \overline{\sigma}, \mathcal{R}[\![\sigma_i]\!]\rangle^{\mathtt{G}}} \quad [\text{SNS}]$$

$$\frac{\overline{\mathtt{l}} = \mathtt{l}_1 \cdots \mathtt{l}_n \qquad (\text{exists } i \in 1..n)\ (\text{for all } j \in i+1..n) \quad \mathtt{l} = \mathtt{l}_i \wedge \mathtt{l} \neq \mathtt{l}_j}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{sense}\ \sigma\ \mathtt{:filter\ l}]\!]\rangle^{\mathtt{G}} \mid \sigma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}\rangle^{\mathtt{Sns}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{v}_i]\!]\rangle^{\mathtt{G}} \mid \sigma = \langle \mathtt{l}_1\,\mathtt{v}_1 \cdots \mathtt{l}_{i-1}\,\mathtt{v}_{i-1}\mathtt{l}_{i+1}\,\mathtt{v}_{i+1} \cdots \mathtt{l}_n\,\mathtt{v}_n\rangle^{\mathtt{Sns}}\end{array}} \quad [\text{PER}]$$

$$\frac{\overline{\mathtt{p}} = \mathtt{p}_1, \ldots, \mathtt{p}_n \qquad \overline{\mathtt{w}} = \mathtt{w}_1 \cdots \mathtt{w}_n \qquad (\text{exists } i \in 1..n) \quad \mathtt{p}_i = \mathtt{p}}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{observe}\ \alpha.\mathtt{p}]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \_, \_\rangle^{\mathtt{A}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{w}_i]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \_, \_\rangle^{\mathtt{A}}\end{array}} \quad [\text{GETA}]$$

$$\begin{array}{c}\gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{focus}(\alpha, \sigma)]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \overline{\sigma}, \_\rangle^{\mathtt{A}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{R}[\![\sigma]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \overline{\sigma}\,\sigma, \_\rangle^{\mathtt{A}}\end{array} \quad [\text{FOC}]$$

$$\begin{array}{c}\gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{unfocus}(\alpha, \sigma)]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \overline{\sigma}, \_\rangle^{\mathtt{A}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{R}[\![\sigma]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \overline{\sigma}\text{-}\sigma, \_\rangle^{\mathtt{A}}\end{array} \quad [\text{UNFOC}]$$

$$\frac{\mathtt{operation\ o}\,(\overline{\mathtt{U}}\,\overline{\mathtt{x}})\ \mathtt{:guard\ e}\ \{\mathtt{e}';\} \in \mathtt{A} \qquad \mathtt{e}_1 = \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] \qquad \mathtt{e}_2 = \mathtt{e}'[\overline{\mathtt{v}}/\overline{\mathtt{x}}]}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{R}[\![\mathtt{use}\ \alpha.o(\overline{\mathtt{v}})\ \mathtt{:sns}\ \sigma]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \_, \overline{\mathtt{O}}\rangle^{\mathtt{A}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{R}[\![\sigma]\!]\rangle^{\mathtt{G}} \mid \alpha = \langle \_, \_, \_, (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\,]\langle \mathtt{e}_1\rangle\{\mathtt{e}_2\})\ \ \overline{\mathtt{O}}\rangle^{\mathtt{A}}\end{array}} \quad [\text{USE}]$$

$$\gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}(\overline{\mathtt{v}})[\,]\{\mathcal{E}[\![\mathtt{fail}]\!]\}]\!])\rangle^{\mathtt{G}} \longrightarrow\ \gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{failed}^{\mathtt{a}}]\!])\rangle^{\mathtt{G}} \quad [\text{FAIL}]$$

$$\frac{\text{if } (\mathtt{a} \in \mathtt{actNames}(\overline{\mathtt{Sr}}) \cup \mathtt{actNames}(\overline{\mathtt{Sr}'})) \text{ then } \mathtt{v} = \mathtt{fls} \text{ else } \mathtt{v} = \mathtt{tr}}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{completed(a)}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{v}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}}\end{array}} \quad [\text{COMPL}]$$

$$\frac{\text{if } (\mathtt{a} \in \mathtt{actStarted}(\overline{\mathtt{Sr}}) \cup \mathtt{actStarted}(\overline{\mathtt{Sr}'})) \text{ then } \mathtt{v} = \mathtt{tr} \text{ else } \mathtt{v} = \mathtt{fls}}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{started(a)}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{v}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}}\end{array}} \quad [\text{STARTED}]$$

$$\frac{\text{if } \ (\mathtt{failed}^{\mathtt{a}} \in \overline{\mathtt{Sr}}\,\overline{\mathtt{Sr}'}) \ \text{ then } \ \mathtt{v} = \mathtt{tr} \ \text{ else } \ \mathtt{v} = \mathtt{fls}}{\begin{array}{c}\gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{failed(a)}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}} \longrightarrow \\ \gamma = \langle \_, \_, \mathcal{S}([\![\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{v}]\!]\ \ \overline{\mathtt{Sr}}'][\mathtt{e}\}]\!])\rangle^{\mathtt{G}}\end{array}} \quad [\text{FAILED}]$$

Figure 9: Reduction rules: agent instance basic instructions

so that the body of `prepare` could be executed. To the configuration on the right-side of the arrow of (5) we can apply rule [SNS] that is

$$\gamma_O = \langle \emptyset, \sigma, \mathcal{R}[\![\texttt{.s}]\!] \rangle^{\texttt{O}} \longrightarrow \gamma_O = \langle \emptyset, \sigma, \mathcal{R}[\![\alpha_{\texttt{C}}]\!] \rangle^{\texttt{O}} \tag{6}$$

where

1. $\mathcal{R}$ is $\texttt{main}(\alpha_{\texttt{C}})[\mathcal{R}_1\ \texttt{S2}]\{\ \}$, and
2. $\mathcal{R}_1$ is $\texttt{prepare}(\alpha_{\texttt{C}})[\ ]\{\texttt{focus}(\alpha_{\texttt{C}}, [\![\ ]\!]);\}$.

Since in the configuration (4) we have the artifact $\alpha_{\texttt{C}}$ we can apply rule [FOC], that is:

$$\begin{array}{l} \alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \emptyset, \emptyset \rangle^{\texttt{C}} \mid \gamma_O = \langle \emptyset, \sigma, \mathcal{R}[\![\texttt{focus}(\alpha_{\texttt{C}}, \sigma]\!] \rangle^{\texttt{O}} \longrightarrow \\ \alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \sigma, \emptyset \rangle^{\texttt{C}} \mid \gamma_O = \langle \emptyset, \sigma, \mathcal{R}[\![\sigma]\!] \rangle^{\texttt{O}} \end{array} \tag{7}$$

where $\mathcal{R}$ is as 1. above, and $\mathcal{R}_1$ is $[\![\ ]\!]$. The effect of [FOC] has been to add the sensor σ private to the agent γ_O in the list of sensors of the artifact $\alpha_{\texttt{C}}$. To γ_O we can apply [END-SA], i.e.,

$$\begin{array}{l} \gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\texttt{prepare}(\alpha_{\texttt{C}})[\ ]\{\sigma\}\ \ \texttt{S2}]\{\ \}]\!]) \rangle^{\texttt{O}} \longrightarrow \\ \gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\ \ \texttt{prepare}(\alpha_{\texttt{C}})\langle \texttt{fls}, \texttt{tr} \rangle\ \ \texttt{S2}\ \ ]\{\ \}]\!]) \rangle^{\texttt{O}} \end{array} \tag{8}$$

As mentioned before, when a sub-activity is completed the rule [END-SA] substitute the sub-activity with a pre-activity, that is the evaluation of the persistence and precondition predicates of the activity. The persistence predicate we use is the one of the declaration of the the sub-activity in the code of the agent, that in this case is `fls`. We can now show an example of the application of rule [DISP] that removes a sub-activity having persistence predicate `fls` from the list of sub-activities (remember that the body of an activity is evaluated only when the list of its sub-activities is empty. This is realized by the first clause of the definition of $\mathcal{R}$). So applying rule [DISP] we have

$$\begin{array}{l} \gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\ \texttt{prepare}(\alpha_{\texttt{C}})\langle \texttt{fls}, \texttt{tr} \rangle\ \ \texttt{S2}\ ]\{\ \}]\!]) \rangle^{\texttt{O}} \longrightarrow \\ \gamma_O = \langle \emptyset, \sigma, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\ \texttt{S2}\ ]\{\ \}]\!]) \rangle^{\texttt{O}} \end{array} \tag{9}$$

Considering the application of the rule [COMPL] to the predicate `completed(prepare)` of `S2` for the configuration on the left side of the

reduction arrow of (9) produces `fls`, whereas for the one on the right side of the arrow produces `tr`.
Assume that we apply rule [SCH] to the agent γ_{U1} of the configuration (4), since both persistence and precondition predicates are `tr`

$$\begin{array}{l}\gamma_{U1} = \langle \emptyset, \sigma_1, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\texttt{SA}_{\texttt{U1}}]\{\ \}]\!])\rangle^{\texttt{U}} \longrightarrow \\ \gamma_{U1} = \langle \emptyset, \sigma_1, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\texttt{usingCount}(\alpha_{\texttt{C}})[\ ]\{\texttt{use c.inc}() : \texttt{sns s};\}]\{\ \}]\!])\rangle^{\texttt{U}}\end{array} \quad (10)$$

where the evaluation context $\mathcal{S}$ is $([\![\ ]\!])$. Now another application of rule [SNS] similar to (6) produces

$$\gamma_{U1} = \langle \emptyset, \sigma_1, \mathcal{S}([\![\texttt{main}(\alpha_{\texttt{C}})[\texttt{usingCount}(\alpha_{\texttt{C}})[\ ]\{\texttt{use c.inc}() : \texttt{sns } \sigma_1;\}]\{\ \}]\!])\rangle^{\texttt{U}}$$

Since in the current configuration we have the artifact $\alpha_{\texttt{C}}$ we can apply rule [USE], that is:

$$\begin{array}{l}\alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \sigma, \emptyset\rangle^{\texttt{C}} \mid \gamma_{U1} = \langle \emptyset, \sigma_1, \mathcal{R}[\![\texttt{use c.inc}() : \texttt{sns } \sigma_1]\!]\rangle^{\texttt{O}} \longrightarrow \\ \alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \sigma, (\sigma_1, \texttt{inc}(\ )[\ ]\langle \texttt{tr}\rangle\{.\texttt{count} = .\texttt{count} + 1\})\rangle^{\texttt{C}} \mid \\ \gamma_{U1} = \langle \emptyset, \sigma, \mathcal{R}[\![\sigma_1]\!]\rangle^{\texttt{U}}\end{array} \quad (11)$$

Note that now the artifact $\alpha_{\texttt{C}}$ contains references to two sensors. One, σ_1, is local to the running operation `inc` and it is used for explicit `signal` expressions that could be evaluated in the body of the operation. The other, σ, is global to the artifact and is used to signal more general events such as updating of properties or end of operations.

In Figure 11 we present the evaluation rules for expressions that may occur only inside operations (and therefore artifacts). Rules [GET] and [SET] return/modify the value of a field. Rule [GETPR] return the value of a property whereas rule [SETPR] sets the value of a property, and moreover, sends the event of updated property to all the agent focusing on the artifact, by inserting in the sensors focusing on the artifact the label $\texttt{prop\_updated}(\texttt{p}_i)$ associated with the value to which the property was set to.

Rule [GEN] signals the event `l` with value `v` by adding the association between `l` and `v` to the sensor associated to the operation, and also to all the sensors of all the agents focusing on the artifact.

Rule [NEXT] generates a new step, adding it to the sequence of steps of the operation. Note that the last two expressions, as well as field and

$$\frac{\begin{array}{l}\texttt{activity a}'\ (\overline{\texttt{T}}\,\overline{\texttt{x}})\ \ \texttt{:agenda}\,(\texttt{SubAct}_1 \cdots \texttt{SubAct}_n)\ \{\texttt{e};\} \in \texttt{G} \\ \texttt{SubAct}_i = \texttt{a}_i(\overline{\texttt{e}})^i\ \ \texttt{:pers e}_i'\ \ \texttt{:pre e}_i''\quad (1 \le i \le n)\end{array}}{\begin{array}{l}\gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}'(\overline{\texttt{v}})\langle \texttt{tr}, \texttt{tr}\rangle\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}} \longrightarrow \\ \quad \gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}'(\overline{\texttt{v}})[\texttt{a}_1(\overline{\texttt{e}}^1[\overline{\texttt{v}}/\overline{\texttt{x}}])\langle \texttt{tr}, \texttt{e}_1''[\overline{\texttt{v}}/\overline{\texttt{x}}]\rangle \cdots \\ \qquad\qquad \cdots \texttt{a}_n(\overline{\texttt{e}}^n[\overline{\texttt{v}}/\overline{\texttt{x}}])\langle \{, \texttt{e}\rangle[\overline{\texttt{v}}/\overline{\texttt{x}}]\}\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}}\end{array}}\ [\text{SCH}]$$

$$\frac{\begin{array}{l}\texttt{activity a}\ (\overline{\texttt{T}}\,\overline{\texttt{x}})\ \ \texttt{:agenda}\,(\texttt{SubAct}_1 \cdots \texttt{SubAct}_n)\ \{\texttt{e};\} \in \texttt{G} \\ \texttt{SubAct}_i = \texttt{a}_i(\overline{\texttt{e}})^i\ \ \texttt{:pers e}_i'\ \ \texttt{:pre e}_i''\quad (1 \le i \le n)\end{array}}{\begin{array}{l}\gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}_i(\overline{\texttt{v}})\langle \texttt{tr}, \texttt{fls}\rangle\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}} \longrightarrow \\ \quad \gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}_i(\overline{\texttt{v}})\langle \texttt{tr}, \texttt{e}_i''[\overline{\texttt{w}}/\overline{\texttt{x}}]\rangle\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}}\end{array}}\ [\text{PREC}]$$

$$\frac{}{\begin{array}{l}\gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}_i(\overline{\texttt{v}})\langle \texttt{fls}, \texttt{e}'\rangle\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}} \longrightarrow \\ \quad \gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'\} \rrbracket)\rangle^{\texttt{G}}\end{array}}\ [\text{DISP}]$$

$$\frac{\begin{array}{l}\texttt{activity a}\ (\overline{\texttt{T}}\,\overline{\texttt{x}})\ \ \texttt{:agenda}\,(\texttt{SubAct}_1 \cdots \texttt{SubAct}_\texttt{n})\ \{\texttt{e};\} \in \texttt{G} \\ \texttt{SubAct}_i = \texttt{a}'(\overline{\texttt{e}})\ \ \texttt{:pers e}'\ \ \texttt{:pre e}''\quad (1 \le i \le n)\end{array}}{\begin{array}{l}\gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}'(\overline{\texttt{v}})[\,]\{\texttt{v}\}\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'''\} \rrbracket)\rangle^{\texttt{G}} \longrightarrow \\ \quad \gamma = \langle \_, \_, \mathcal{S}(\llbracket \texttt{a}(\overline{\texttt{w}})[\overline{\texttt{Sr}}\ \ \texttt{a}'(\overline{\texttt{e}}[\overline{\texttt{w}}/\overline{\texttt{x}}])\langle \texttt{e}'[\overline{\texttt{w}}/\overline{\texttt{x}}], \texttt{e}''[\overline{\texttt{w}}/\overline{\texttt{x}}]\rangle\ \ \overline{\texttt{Sr}}'] \{\texttt{e}'''\} \rrbracket)\rangle^{\texttt{G}}\end{array}}\ [\text{END-SA}]$$

Figure 10: Reduction rules: agent instance scheduling of sub-activities

property update may only occur in the evaluation of the body of an operation (and not in the evaluation of its guard that is side effect free).

Assume that we schedule the execution of the operation `inc` of $\alpha_{\texttt{C}}$. First, as the evaluation context for expressions require, we have to evaluate the right-hand-side of the assignment to the property `count`. Even though, we do not have arithmetical operations in our definition of expressions, we can assume that they are evaluated by first evaluating their arguments and them returning the result of the arithmetical operation. So

$$\alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \sigma, (\sigma_1, \texttt{inc}(\,)[\,]\langle \texttt{tr}\rangle\{.\texttt{count} = .\texttt{count} + 1\})\rangle^{\texttt{C}}$$

can be written as

$$\alpha_{\texttt{C}} = \langle \emptyset, \texttt{count} = 0, \sigma, \mathcal{O}[\![.\texttt{count}]\!]\rangle^{\texttt{C}}$$

where $\mathcal{O}$ is $(\sigma_1, \texttt{inc}(\ )[\ ]\langle\texttt{tr}\rangle\{.\texttt{count} = [\![\ ]\!] + 1\})$. We can apply rule [GETPR] that is:

$$\alpha_{\texttt{c}} = \langle\emptyset, \texttt{count} = 0, \sigma, \mathcal{O}[\![.\texttt{count}]\!]\rangle^{\texttt{c}} \longrightarrow \alpha_{\texttt{c}} = \langle\emptyset, \texttt{count} = 0, \sigma, \mathcal{O}[\![0]\!]\rangle^{\texttt{c}} \tag{12}$$

Assume that the evaluation of the expression $0 + 1$ produces 1, the artifact can be written as

$$\alpha_{\texttt{c}} = \langle\emptyset, \texttt{count} = 0, \sigma, (\sigma_1, \texttt{inc}(\ )[\ ]\langle\texttt{tr}\rangle\{\mathcal{E}[\![.\texttt{count} = 1]\!]\})\rangle^{\texttt{c}}$$

where $\mathcal{E}$ is $[\![\ ]\!]$. The application of rule [SETPR] is as follows:

$$\begin{array}{l}\alpha_{\texttt{c}} = \langle\emptyset, \texttt{count} = 0, \sigma, (\sigma_1, \texttt{inc}(\ )[\ ]\langle\texttt{tr}\rangle\{\mathcal{E}[\![.\texttt{count} = 1]\!]\})\rangle^{\texttt{c}} \mid \sigma = \langle\emptyset\rangle^{\texttt{Sns}} \longrightarrow \\ \alpha_{\texttt{c}} = \langle\emptyset, \texttt{count} = 1, \sigma, (\sigma_1, \texttt{inc}(\ )[\ ]\langle\texttt{tr}\rangle\{1\})\rangle^{\texttt{c}} \mid \sigma = \langle\texttt{l}\ 1\rangle^{\texttt{Sns}}\end{array} \tag{13}$$

where `l` is `prop_updated(count)`. So now if the `Observer` agent that focused on the artifact is sensing the change of property `count`, it would get as result the new value of the property.

In Figure 12 we present the state change rules for artifacts. Note that, given the definition of $\mathcal{K}$, and the rules in Figure 11 the operation that is evaluated is always the last of the sequence (first of the queue). When the guard of an operation is `tr` and the body is fully evaluated, there are two cases: either some step was generated (during the evaluation of the body), in which case rule [SELG] removes the current operation from the queue and enqueues the step evaluation expression of the sequence of steps; or no step was generated, that is, the operation is completed and can be dequeued, rule [END]. In addition, rule [END] signals to all the agents focusing on the artifact the completion of the operation. Since there is no value associated with this event we use the value `unit`.

When the guard of an operation evaluates to `fls`, with rule [FLS], the operation with its guard is dequeued and again enqueued in the queue of operations (so, when its turn comes, it will be reevaluated) Note that, no step could be generated during its evaluation since the guard is side effect free.

The last two rules are applied when the operation in execution is the evaluation of the guards of operation steps. If a guard of a step evaluates to `tr`, then the evaluation continues with the body of the step, rule [SGO],

$$\frac{\overline{\mathtt{U}}\ \overline{\mathtt{f}} \in \mathtt{A} \qquad \mathtt{f}_i \in \overline{\mathtt{f}}}{\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \_, \_, \overline{\mathtt{O}} \ \ \mathcal{O}[\![.\mathtt{f}_i]\!] \rangle^{\mathtt{A}} \ \longrightarrow \ \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \_, \_, \overline{\mathtt{O}} \ \ \mathcal{O}[\![\mathtt{v}_i]\!] \rangle^{\mathtt{A}}} \quad [\text{GET}]$$

$$\frac{\overline{\mathtt{U}}\ \overline{\mathtt{f}} \in \mathtt{A} \qquad \mathtt{f}_i \in \overline{\mathtt{f}} \qquad \overline{\mathtt{w}} = \overline{\mathtt{v}}[i \mapsto \mathtt{v}]}{\begin{array}{c}\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \_, \_, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![.\mathtt{f}_i = \mathtt{v}]\!]\}) \rangle^{\mathtt{A}} \ \longrightarrow \\ \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{w}}, \_, \_, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{v}]\!]\}) \rangle^{\mathtt{A}}\end{array}} \quad [\text{SET}]$$

$$\frac{\mathtt{p}_i\ \mathtt{w}_i \in \overline{\mathtt{p}}\ \overline{\mathtt{w}}}{\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \_, \_, \overline{\mathtt{O}} \ \ \mathcal{O}[\![.\mathtt{p}_i]\!] \rangle^{\mathtt{A}} \ \longrightarrow \ \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \_, \_, \overline{\mathtt{O}} \ \ \mathcal{O}[\![\mathtt{w}_i]\!] \rangle^{\mathtt{A}}} \quad [\text{GETPR}]$$

$$\frac{\overline{\sigma} = \sigma_1 \cdots \sigma_n \qquad \mathtt{p}_i\ \mathtt{v}_i \in \overline{\mathtt{p}}\ \overline{\mathtt{v}} \qquad \overline{\mathtt{w}} = \overline{\mathtt{v}}[i \mapsto \mathtt{v}] \qquad \mathtt{l} = \mathtt{prop\_updated}(\mathtt{p}_i)}{\begin{array}{l}\alpha = \langle \_, \overline{\mathtt{p}} = \overline{\mathtt{v}}, \overline{\sigma}, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![.\mathtt{p}_i = \mathtt{v}]\!]\}) \rangle^{\mathtt{A}} \\ \mid \sigma_1 = \langle \overline{\mathtt{l}}^1\ \overline{\mathtt{v}}^1 \rangle^{\mathtt{Sns}} \mid \cdots \mid \sigma_n = \langle \overline{\mathtt{l}}^n\ \overline{\mathtt{v}}^n \rangle^{\mathtt{Sns}} \\ \longrightarrow \\ \alpha = \langle \_, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{v}]\!]\}) \rangle^{\mathtt{A}} \\ \mid \sigma_1 = \langle \mathtt{l}\ \mathtt{v} \ \ \overline{\mathtt{l}}^1\ \overline{\mathtt{v}}^1 \rangle^{\mathtt{Sns}} \mid \cdots \mid \sigma_n = \langle \mathtt{l}\ \mathtt{v} \ \ \overline{\mathtt{l}}^n\ \overline{\mathtt{v}}^n \rangle^{\mathtt{Sns}}\end{array}} \quad [\text{SETPR}]$$

$$\frac{\overline{\sigma} = \sigma_1 \cdots \sigma_n}{\begin{array}{l}\alpha = \langle \_, \_, \overline{\sigma}, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{signal}(\mathtt{l}(\mathtt{v}))]\!]\}) \rangle^{\mathtt{A}} \\ \mid \sigma = \langle \overline{\mathtt{l}}\ \overline{\mathtt{v}} \rangle^{\mathtt{Sns}} \mid \sigma_1 = \langle \overline{\mathtt{l}}^1\ \overline{\mathtt{v}}^1 \rangle^{\mathtt{Sns}} \mid \cdots \mid \sigma_n = \langle \overline{\mathtt{l}}^n\ \overline{\mathtt{v}}^n \rangle^{\mathtt{Sns}} \\ \longrightarrow \\ \alpha = \langle \_, \_, \overline{\sigma}, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{v}]\!]\}) \rangle^{\mathtt{A}} \\ \mid \sigma = \langle \mathtt{l}\ \mathtt{v} \ \ \overline{\mathtt{l}}\ \overline{\mathtt{v}} \rangle^{\mathtt{Sns}} \mid \sigma_1 = \langle \mathtt{l}\ \mathtt{v} \ \ \overline{\mathtt{l}}^1\ \overline{\mathtt{v}}^1 \rangle^{\mathtt{Sns}} \mid \cdots \mid \sigma_n = \langle \mathtt{l}\ \mathtt{v} \ \ \overline{\mathtt{l}}^n\ \overline{\mathtt{v}}^n \rangle^{\mathtt{Sns}}\end{array}} \quad [\text{GEN}]$$

$$\frac{\mathtt{step\ o'}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \mathtt{:guard\ e}\ \{\cdots\} \in \mathtt{A}}{\begin{array}{c}\alpha = \langle \_, \_, \_, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{next\ o'}(\overline{\mathtt{w}})]\!]\}) \rangle^{\mathtt{A}} \ \longrightarrow \\ \alpha = \langle \_, \_, \_, \overline{\mathtt{O}} \ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}} \ \ \mathtt{o'}(\overline{\mathtt{w}})\langle \mathtt{e}[\overline{\mathtt{w}}/\overline{\mathtt{x}}] \rangle] \langle \mathtt{tr} \rangle \{\mathcal{E}[\![\alpha]\!]\}) \rangle^{\mathtt{A}}\end{array}} \quad [\text{NEXT}]$$

Figure 11: Reduction rules: artifact instance basic instructions

and everything else is disregarded. If instead all the guards of the steps evaluated to `fls`, with rule [SKIP] their evaluation is rescheduled and put at the beginning of the sequence.

An application of rule [END] is as follows:

$$\begin{array}{l}\alpha_{\mathtt{C}} = \langle \emptyset, \mathtt{count} = 1, \sigma, (\sigma_1, \mathtt{inc}(\ )[\ ]\langle \mathtt{tr} \rangle \{1\}) \rangle^{\mathtt{C}} \mid \sigma = \langle \mathtt{l}\ 1 \rangle^{\mathtt{Sns}} \longrightarrow \\ \alpha_{\mathtt{C}} = \langle \emptyset, \mathtt{count} = 1, \sigma, \emptyset \rangle^{\mathtt{C}} \mid \sigma = \langle \mathtt{l'\ unit\ l}\ 1 \rangle^{\mathtt{Sns}}\end{array} \tag{14}$$

where $\mathtt{l}'$ is `op_exec_completed(inc)`. This event could be sensed by the `Observer` agent that focused on the artifact.

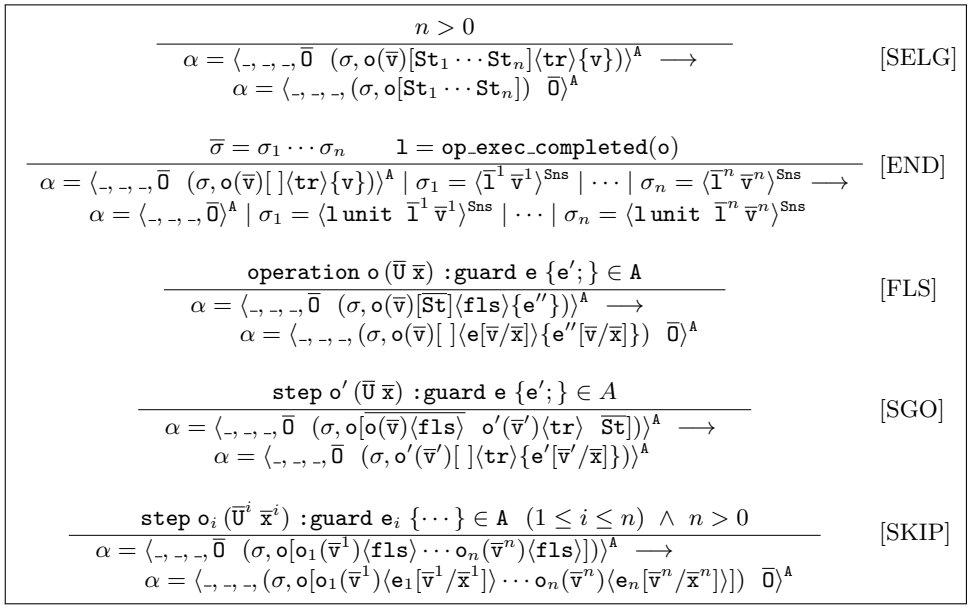

Figure 12: Reduction rules: artifact instance state change

4 FAAL Type Soundness

This section shows that FAAL enjoys the standard *type soundness* property of statically typed languages. Namely, the execution of a well-typed program does not get stuck: that is, if a running agent has some ongoing activity or an artifact has some operation to perform, then there is some rule that can be applied. To state the type soundness result for FAAL we introduce a suitable notion of typing for configurations, show that the initial configuration of a well-typed program is well-typed, and that reducing a well-typed configuration produces a well typed configuration (subject reduction). Moreover, we show that if an agent it is not sensing an event or it does not have failed sub-activities, then some rule is applicable (progress). In this section we only state the main results, whose (technical) proof is given in the Appendix.

4.1 Well-Typed Configurations

In order to give types to run-time expressions we have to modify the type system by replacing the type environment Γ (mapping variables to

$$\frac{\texttt{activity a}(\overline{\texttt{T}}\,\overline{\texttt{x}})\ \cdots \in \texttt{G} \quad \Sigma \Vdash \texttt{e} : \texttt{Bool IN G} \quad \Sigma \Vdash \texttt{e}' : \texttt{Bool IN G} \quad \Sigma \Vdash \overline{\texttt{e}} : \overline{\texttt{T}} \text{ IN G}}{\Sigma \Vdash \texttt{a}(\overline{\texttt{e}})\langle \texttt{e}, \texttt{e}' \rangle \text{ OK IN G}}$$

$$\frac{\texttt{activity a}(\overline{\texttt{T}}\,\overline{\texttt{x}})\ \cdots \in \texttt{G} \quad (\text{for all } i \in 1..n) \quad \Sigma \Vdash \texttt{Sr}_i \text{ OK IN G} \quad \Sigma \Vdash \texttt{e} : \cdots \text{ IN G} \quad \Sigma \Vdash \overline{\texttt{v}} : \overline{\texttt{T}} \text{ IN G}}{\Sigma \Vdash \texttt{a}(\overline{\texttt{v}})[\texttt{Sr}_1 \cdots \texttt{Sr}_n]\{\texttt{e}\} \text{ OK IN G}}$$

$$\frac{\texttt{activity a}\ \cdots \in \texttt{G}}{\Sigma \Vdash \texttt{failed}^{\texttt{a}} \text{ OK IN G}}$$

$$\frac{\Sigma \Vdash \texttt{R OK IN G} \quad \texttt{agent G}\,\{\,\texttt{Sns}\ \texttt{s}_1 \cdots \texttt{s}_n;\ \cdots\} \quad (\text{for all } \sigma \in \sigma_1 \cdots \sigma_n) \quad \Sigma(\sigma) = \texttt{Sns}}{\Sigma \Vdash \gamma = \langle \overline{\texttt{l}}\,\overline{\texttt{v}}, \overline{\sigma}, \texttt{R} \rangle^{\texttt{G}} \text{ OK}}$$

Figure 13: Well typed run-time sub-activities/activities and agent instances

types) with the run-time type environment Σ, denoted by $[\overline{\iota} : \overline{\texttt{T}}]$, mapping agents/artifacts/sensors references to types. The new typing judgement

$$\Sigma \Vdash \texttt{e} : \texttt{T IN X}$$

means that, under the assumptions in Σ for references, the expression e has type T in the context of an instance of the artifact or agent X. The rules of the system are obtained from the rules of Figure 4 by replacing Γ with Σ, and replacing rule [Tvar], with

$$\Sigma \Vdash \iota : \Sigma(\iota) \text{ IN X} \quad [\text{Tid}]$$

The typing of run-time expressions is used to define well typed configurations. In particular the judgments for:

- run-time sub-activities/activities and agent instances are given in Figure 13;
- run-time steps/operations and artifact instances are given in Figure 14;
- run-time sensor instances are given in Figure 15;
- configurations are given in Figure 16, where the auxiliary function typeEnv is defined by:

$$\texttt{typeEnv}(\iota_1 = \langle \cdots \rangle^{\texttt{T}_1} \mid \cdots \mid \iota_n = \langle \cdots \rangle^{\texttt{T}_n}) = [\iota_1 : \texttt{T}_1, \ldots, \iota_n : \texttt{T}_n]$$

Note that if $\vdash \texttt{M}$ OK, and $\texttt{M} \equiv \texttt{M}'$, then $\vdash \texttt{M}'$ OK.

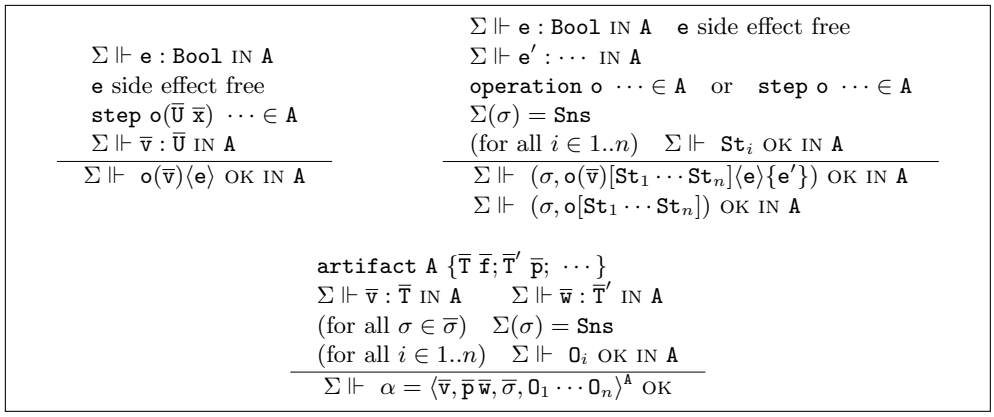

Figure 14: Well typed run-time steps/operations and artifact instances

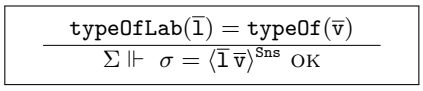

Figure 15: Well typed run-time sensor instances

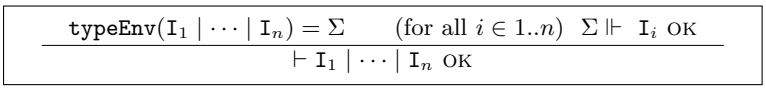

Figure 16: Well typed configurations

4.2 Type Soundness

In this section we state Subject Reduction and Progress for well-typed configurations, and Type Soundness of well-typed programs. In the following we assume that the program containing the definition of the agents and artifacts is well-typed, i.e., $\vdash (\mathtt{GT}, \mathtt{AT})$ OK.

The subject reduction theorem says that by reducing a well typed configuration we obtain a well typed configuration.

Theorem 1 (Subject Reduction) *If* $\vdash \mathtt{M}$ OK *and* $\mathtt{M} \Rightarrow \mathtt{M}'$, *then* $\vdash \mathtt{M}'$ OK.

Proof: The proof is given in Appendix A.1. □

In order to state the progress result we define when an agent has completed its task, successfully or with failure, and when *an agent is blocked* in a configuration, that is the agent cannot reduce. This is if all its activities are doing a sense on sensors not containing the specified label (so to proceed in the evaluation of the activity the agent has to wait for some artifact to signal on those sensors).

Definition 3 (Completed agent) *The agent instance* $\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ *is* completed *if* $\mathtt{R} = \mathtt{failed}^{\mathtt{main}}$, *or* $\mathtt{R} = \mathtt{main}(\overline{\mathtt{v}})[\ ]\{\mathtt{v}\}$.

Definition 4 (Blocked agent in a configuration) *Let* $\mathtt{M} = \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ *be a configuration.*

- *The* expression $\mathtt{e}$ is blocked in $\mathtt{M}$ *if* $\mathtt{e} = \mathcal{E}[\![\mathtt{sense}\ \sigma\ \mathtt{:filter\ l}]\!]$, *and there exist* $j \in 1..n$ *such that* $\mathtt{I}_j$ *is* $\sigma = \langle \overline{\mathtt{l}}'\,\overline{\mathtt{v}}'\rangle^{\mathtt{Sns}}$ *and* $\mathtt{l} \notin \overline{\mathtt{l}}'$.
- *The* activity $\mathtt{a}(\overline{\mathtt{v}})[\ \mathtt{Sr}_1 \cdots \mathtt{Sr}_m\ ]\{\mathtt{e}\}$ is blocked in $\mathtt{M}$ *if*
  1. *either* $m = 0$, *and* $\mathtt{e}$ *is blocked in* $\mathtt{M}$, *or*
  2. $m > 0$, *and for all* i, $1 \leq i \leq m$, *either* $\mathtt{Sr}_i$ *is is blocked in* $\mathtt{M}$, *or* $\mathtt{Sr}_i = \mathtt{failed}^{\mathtt{a}'}$ *for some* $\mathtt{a}'$.
- *The* agent $\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ is blocked in $\mathtt{M}$ *if* $\mathtt{R}$ *is blocked in* $\mathtt{M}$.

Note that in the definition of blocked activity we require that the sensor on which the activity is doing the sense be defined. The type system will ensure that the required sensor is present. Obviously, absence or presence of the label is used to coordinate agents and artifacts.

An artifact is idle if it does not have any pending operation.

Definition 5 (Idle artifact instance) *An artifact instance* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}} \rangle^{\mathtt{A}}$ *is* idle *if* $\overline{\mathtt{O}} = \emptyset$.

Theorem 2 (Progress) *If* $\vdash \mathtt{M}$ OK, $\mathtt{M} = \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ *and (for some* $i \in 1..n$*)* $\mathtt{I}_i$ *is an agent non-completed and non-blocked in* M*, or a non-idle artifact, then* $\mathtt{M} \Rightarrow \mathtt{M}'$ *for some* $\mathtt{M}'$.

Proof: The proof is given in Appendix A.2. □

Theorem 3 (Type Soundness) *If* I *is the initial configuration of* (GT, AT) *and* $\mathtt{I} \Rightarrow^{\star} \mathtt{M}$ *with* $\mathtt{M} = \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ *not reducible, then* $\vdash \mathtt{M}$ OK *and (for all* $i \in 1..n$*)* $\mathtt{I}_i$ *is either a sensor or an idle artifact or a completed or blocked agent.*

Proof: First observe that, if I is the initial configuration of (GT, AT), then $\vdash \mathtt{I}$ OK (the proof is straightforward by induction on expressions). Then, the result follows immediately from Theorems 1 and 2. □

5 Related Work

The literature related to the present paper has been partially quoted in the introduction. We add here comparisons and remarks concerning core calculi for agents, actors and concurrent objects.

In the Agent-Oriented Programming literature, many contributions introduced a formal semantics for abstract or concrete agent programming languages, with the purpose of providing a rigorous and formal account for the design, specification and verification of agent programs. One of the first examples is [29], which describes the operational semantics of an abstract BDI-based agent programming language, combining features of logic programming and imperative programming. Actually, operational semantics has been widely adopted to formally describe the behaviour of agent programming languages and frameworks (e.g., [50, 57]), as well as of specific aspects of multi-agent systems, such as agent communication (e.g., [46, 22]), agent organisations (e.g., [23]), etc. Examples of concrete agent programming languages which enjoy a formal operational semantics are *Jason* [9], 2APL [19] and GOAL [30].

In all these works, the operational semantics is used essentially to specify formally the behaviour of the agent (abstract) machines executing agent programs. To the authors' knowledge, no investigations about a type

system supporting standard type soundness for an agent-based programming framework have been developed so far. The most direct attempt that has been done so far applying formal modelling techniques like core calculi to study properties of agent-oriented programs and of agent-oriented extensions of object-oriented systems is [52], to which the present work is mainly an extension. Such a formalisation, however, lacks a type system that is able to guarantee well-formedness properties of programs. Building on top of [52], this paper formalised a larger set of features (including *agent agenda* and *artifact properties*) and provided a type soundness result. In [17, 18] we introduced a core calculus for agents and artifacts (of which the current calculus is an extension), outlined its operational semantics, and discussed its main properties.

Core calculi and type systems are applied to programming frameworks which, although not strictly agent-based, have many similarities that are worth mentioning. In [11, 44] a calculus is introduced on a very weak notion of agent, namely, an active entity (like an actor or process) exchanging messages asynchronously. Accordingly, such a formalisation does not consider the *environment* and related concepts, concerning percepts and actions, which are instead a core part of FAAL. These aspects are considered instead in the layered agent calculus [38], which is not introduced – however – for formalising the features of an agent programming language.

In their seminal book [1], Abadi and Cardelli develop a theory of objects as a foundation for object-oriented languages and programming, and introduce some object calculi, which are formalisms at the same level of abstraction as function calculi, but based exclusively on objects rather than functions. They study both functional/imperative and untyped/typed systems. Our modelling of artifacts, which can be seen as asyncronous objects, is inspired to their imperative calculi. However, concurrency, situatedness and autonomy, are not considered, and therefore agents, cannot be straightforwardly modelled with such calculi.

A number of core languages have been proposed for actors and object-oriented concurrent programming, almost all of which are based on some kind of process calculi. Examples are: CAP [12], a process calculus based on the actor model; Honda and Tokoro's object calculus with asynchronous communication [31]; the Join calculus [24], which has been used in the formalisation of features of various concurrent programming languages (such as Polyphonic C# [6] and JOIN JAVA [36]); and STATEJ [16, 15] (see also [14, 13]), that proposes state classes, a construct for making the state of a

concurrent object explicit.

Type systems have been widely used for analysing the behavior of concurrent programs and systems of concurrent processes, to reason about deadlock-freedom, safe usage of locks, etc. [64, 41]. Typically, variants of the π-calculus [45] are used as target language. Some process calculi extending the π-calculus have been introduced to model specifically *mobile agents* [25, 48, 20]. Nomadic Pict, see [55], emphasizes location dependence/independence, and provides a distributed infrastructure, see [56], for the migration of mobile agents. The Distributed-π calculus [28, 27] is a typed language for mobile agents extending the π-calculus with an explicit notion of location that represent the environment where such agents are currently located. The type system is based on the notion of *location types*, which describe the set of resources (i.e., typed channels to communicate with other agents) available to an agent at a location. The notion of type in this case is introduced for controlling the use of resources in a distributed system.

Finally, the notion of *session type* has been introduced to specify complex interaction protocols, verified by static typechecking [32]. In the core calculus presented the interaction between agents and artifacts, and the dependency between actions and sub-actions is programmed via predicates: preconditions for activities and guards for artifacts. Session types, and in particular multiparty session types (see [33]), could be used to impose (and verify statically) restrictions on the pattern of interaction, as hinted in the following section.

6 Conclusion and Further Work

The FAAL calculus provides a first step towards a rigorous formal framework for designing agent-oriented languages and studying properties of agent-oriented programs. It enjoys the standard type soundness property of statically typed languages. Namely, the execution of a well-typed program does not get stuck: that is, if a running agent has some ongoing activity or an artifact has some operation to perform, then there is some rule that can be applied. In particular, the type system has been designed to guarantee that a number of properties are satisfied, including the following: *(i)* agents may execute (and query) activities and access sensors only if these are defined for them; *(ii)* agents may invoke operations and observe properties only if these are defined for the target artifact; *(iii)* artifacts may only access/modify

fields and properties defined for them; and finally, *(iv)* channels (sensors) supporting the communication between agents and artifacts are typed so that they signal values of known types.

Future work will be focussed on two main directions. The first direction concerns enriching the current language and its formal model. In particular, 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 the set of activities, such as the fact that an activity A would be executed always after an activity A' or that activities A and A' cannot be executed together. The second direction is about studying and defining agent calculi and type systems for agent oriented programming languages based on high-level models/architectures, such as BDI (Belief Desire Intention) [49], which is one of the main references in programming rational/intelligent agents. Differently from SIMPA, such languages adopt high-level *cognitive* concepts – such as tasks, goals, plans, beliefs – to define agent structure and behaviour. A medium-term goal we believe can be reached is to substantially fill the gap between object- and agent-orientation, fostering the adoption of new metaphors, abstractions and patterns for tackling the concurrency issue.

Acknowledgements

The insightful comment and suggestions that we received during the review process have been of great help in improving the paper.

APPENDIX

A Proofs

A.1 Proof of Theorem 1 (Subject Reduction)

Both the type system for programs (system $\vdash$, presented in Section 3.2) and the type system for configurations (system $\Vdash$, presented in Section 4.1) are syntax directed and enjoy the inversion property. The inversion property for the typing rule for configurations (in Figure 16) is given by Lemma 1.1 below. We do not give the inversion lemmas for the other rules of systems $\vdash$ and $\Vdash$ since they are trivial. Let X range over agent/artifact types and let `Y` range over agent/artifact types and `C`.

Lemma 1 *1. If* $\vdash \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ OK *and* $\mathtt{typeEnv}(\mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n) = \Sigma$*, then* $\Sigma \Vdash \mathtt{I}_i$ OK *(for all* $i \in 1..n$*).*

2. Let $\Sigma \Vdash \mathtt{I}$ OK*, then there exists a type* $\mathtt{T}$ *such that*

(a) $\mathtt{I} = (\gamma = \langle \overline{\mathtt{l}}\ \overline{\mathtt{v}}, \overline{\sigma}, \mathcal{R}[\mathtt{e}]\rangle^{\mathtt{G}})$ *implies* $\Sigma \Vdash \mathtt{e} : \mathtt{T}$ IN $\mathtt{G}$*, and*

(b) $\mathtt{I} = (\alpha = \langle \overline{\mathtt{v}}, \overline{\mathtt{p}}\ \overline{\mathtt{w}}, \overline{\sigma},\ \overline{\mathtt{O}}\ \mathcal{O}[\mathtt{e}]\rangle^{\mathtt{A}})$ *implies* $\Sigma \Vdash \mathtt{e} : \mathtt{T}$ IN $\mathtt{A}$*.*

Proof:

1. By definition of well typed configurations, see Figure 16
2. By structural induction on the contexts $\mathcal{R}$ and $\mathcal{O}$ (see Section 3.3.3) using the typing rules in Figures 4, 5 and 6.

□

Lemma 2 (Weakening) *If* $\Sigma \Vdash \mathtt{I}$ OK *and* $\Sigma' \supseteq \Sigma$*, then* $\Sigma' \Vdash \mathtt{I}$ OK*.*

Proof: Straightforward, by induction on derivations. □

To simplify the presentation, we define a context $\mathcal{G}$ that (non deterministically) selects a sub-activity in an agent instance (the context $\mathcal{S}$ has been defined in Section 3.3).

$$\mathcal{G} \ ::= \ \gamma = \langle \overline{\mathtt{l}}\ \overline{\mathtt{v}}, \overline{\sigma}, \mathcal{S}\rangle^{\mathtt{G}}$$

Lemma 3 (Replacement) *1. Let* $\Sigma \Vdash \mathcal{K}[\mathtt{e}]$ OK *and* $\Sigma \Vdash \mathtt{e} : \mathtt{T}$ IN $\mathtt{X}$*, then* $\Sigma \Vdash \mathtt{e}' : \mathtt{T}$ IN $\mathtt{X}$ *implies* $\Sigma \Vdash \mathcal{K}[\mathtt{e}']$ OK*.*

2. Let $\Sigma \Vdash \mathcal{G}[\mathtt{Sr}]$ OK *and* $\Sigma \Vdash \mathtt{Sr}$ OK IN $\mathtt{G}$*, then* $\Sigma \Vdash \mathtt{Sr}'$ OK IN $\mathtt{G}$ *implies* $\Sigma \Vdash \mathcal{G}[\mathtt{Sr}']$ OK*.*

Proof: Straightforward, by induction on derivations. □

Lemma 4 (Substitution) *1. If* $[\overline{\mathtt{x}}\ \overline{\mathtt{x}}' : \overline{\mathtt{X}}\ \overline{\mathtt{C}}] \vdash \mathtt{e} : \mathtt{T}$ IN $\mathtt{G}$*, and* $\mathtt{typeOf}(\overline{\mathtt{v}}) = \overline{\mathtt{C}}$*, then* $[\overline{\iota} : \overline{\mathtt{X}}] \Vdash \mathtt{e}[\overline{\iota}\ \overline{\mathtt{v}}/\overline{\mathtt{x}}\ \overline{\mathtt{x}}'] : \mathtt{T}$ IN $\mathtt{G}$*.*

2. If $[\overline{\mathtt{x}}\ \overline{\mathtt{x}}' : \overline{\mathtt{Y}}\ \overline{\mathtt{C}}] \vdash \mathtt{e} : \mathtt{T}$ IN $\mathtt{A}$*, and* $\mathtt{typeOf}(\overline{\mathtt{v}}) = \overline{\mathtt{C}}$*, then* $[\overline{\iota} : \overline{\mathtt{Y}}] \Vdash \mathtt{e}[\overline{\iota}\ \overline{\mathtt{v}}/\overline{\mathtt{x}}\ \overline{\mathtt{x}}'] : \mathtt{T}$ IN $\mathtt{A}$*.*

Proof: Straightforward, by induction on expressions. □

Lemma 5 (Subject Congruence) *If* $\vdash \mathtt{M}$ OK *and* $\mathtt{M} \equiv \mathtt{M}'$*, then* $\vdash \mathtt{M}'$ OK*.*

Proof: Straightforward, by using Lemma 1.1. □

Lemma 6 (Subject Reduction) *1. If* $\vdash \mathtt{M}_0$ OK *and* $\mathtt{M}_0 \longrightarrow \mathtt{M}_1$*, then* $\vdash \mathtt{M}_1$ OK*.*

2. If $\vdash \mathtt{M}' \mid \mathtt{M}_0$ OK *and* $\mathtt{M}_0 \longrightarrow \mathtt{M}_1$*, then* $\vdash \mathtt{M}' \mid \mathtt{M}_1$ OK*.*

Proof: We consider the proof of 1. (the proof of 2. is similar). The proof is by case analysis on the operational semantics rule (in Figures 8, 9, 10, 11 or 12) used. Cases [SEQ], [MMmatch] (only first conclusion), [MMmismatch], [SNS], [GETA], [COMPL], [STARTED], [FAILED], [GET] and [GETPR] are immediate by Lemma 3.1. Cases [FAIL] and [DISP] are immediate by Lemma 3.2.

Case [AGN]. We have $\Sigma \Vdash \mathcal{K}[\![\mathtt{spawn}\ G(\overline{\mathtt{v}})]\!]$ OK. Let $\Sigma' = \Sigma \cup [\gamma = \mathtt{G}, \sigma_1 : \mathtt{Sns}, \ldots, \sigma_m : \mathtt{Sns}]$. By Lemmas 2 and 3.1 we have $\Sigma' \Vdash \mathcal{K}[\![\gamma]\!]$ OK. From $\vdash \mathtt{agent\ G}\ \{\cdots\}$ OK, by inversion for $\vdash$, Lemma 4.1 and the typing rules in Figure 13 we get $\Sigma' \Vdash \gamma = \langle \emptyset, \overline{\sigma}, \mathtt{main}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_n]\{\mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\}\rangle^{\mathtt{G}}$ OK. By the typing rule in Figure 15 we get $\Sigma' \Vdash \sigma_i = \langle \emptyset \rangle^{\mathtt{Sns}}$ OK $(1 \geq i \geq m)$. Then, by the tying rule in Figure 16 we get $\Sigma' \Vdash \mathcal{K}[\![\gamma]\!] \mid \gamma = \langle \emptyset, \overline{\sigma}, \mathtt{main}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_n]\{\mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\}\rangle^{\mathtt{G}} \mid \sigma_1 = \langle \emptyset \rangle^{\mathtt{Sns}} \mid \cdots \mid \sigma_m = \langle \emptyset \rangle^{\mathtt{Sns}}$ OK.

Case [ART]. Similar to the previous case.

Cases [MMmatch] (second and third conclusion), [MMins], [PER], [FOC] and [UNFOC]. Straightforward by using Lemmas 1 and 3.1.

Case [USE]. Straightforward by using Lemmas 1, 3.1 and 4.2.

Cases [SCH], [PREC] and [END-SA]. Straightforward by using Lemma 3.2 and 4.1.

Cases [SET], [SETPR], [GEN], [SELG] and [END]. Straightforward by using inversion.

Cases [NEXT], [FLS], [SGO] and [SKIP]. Straightforward by using inversion and Lemma 4.2. □

Proof of Theorem 1 (Subject Reduction):

By case analysis on the operational semantics rule (in Figure 7) used. Case [EMPTYCONT] follows by Lemmas 5 and 6.1. Case [CONT] follows by Lemmas 5 and 6.2.

A.2 Proof of Theorem 2 (Progress)

Let the set of expressions be partitioned in the following sets:

$$\begin{array}{rcl}
\mathcal{E}_{\mathtt{X}} & = & \{\mathtt{spawn\,G}(\overline{\mathtt{e}}), \mathtt{make\,A}(\overline{\mathtt{e}}), \mathtt{e}_1; \mathtt{e}_2, \mathtt{v}\} \\
\mathcal{E}_{\mathtt{A}} & = & \{\mathtt{.f}, \mathtt{.f = e}, \mathtt{.p}, \mathtt{.p = e}, \mathtt{next\ o}(\overline{\mathtt{e}}), \mathtt{signal(l(e))}\} \\
\mathcal{E}_{\mathtt{G}} & = & \{\mathtt{.s}, \mathtt{use\ e_2.o}(\overline{\mathtt{e}})\ \mathtt{:sns\ e_1}, \mathtt{sense\ e\ :filter\ l}, \\
 & & \mathtt{?memo(l)}, \mathtt{-memo(l)}, \mathtt{+memo(l(e))}, \mathtt{memo(l)}, \\
 & & \mathtt{focus}(\mathtt{e}_1, \mathtt{e}_2), \mathtt{unfocus}(\mathtt{e}_1, \mathtt{e}_2), \mathtt{observe\ e.p}, \\
 & & \mathtt{completed(a)}, \mathtt{failed(a)}, \mathtt{fail}\}
\end{array}$$

Lemma 7 *Let* $\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ *be an agent instance, and let* $\Sigma \Vdash \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ OK *for some* Σ*, and* $\vdash$ `agent G { Sns` $\overline{\mathtt{s}}$`;` $\overline{\mathtt{Act}}$`}` OK*. Then* R *contains only (sub)expressions in* $\mathcal{E}_{\mathtt{G}} \cup \mathcal{E}_{\mathtt{X}}$*, or identifiers in* $\overline{\mathtt{s}}$*, i.e.,* R *does not contain variables, or identifiers not in* $\overline{\mathtt{s}}$*, or expressions in* $\mathcal{E}_{\mathtt{A}}$*.*

Proof: Observe that the typing rules of Figure 4 and 5 are such that only expressions in $\mathcal{E}_{\mathtt{G}} \cup \mathcal{E}_{\mathtt{X}}$ are allowed in activities, and only sensor identifiers in $\overline{\mathtt{s}}$. The rule for agent creation, [AGN] of Figure 8, replaces all the variables in the body of the `main` activity of the agent with values, and the reduction rules of Figure 9 replace expressions in $\mathcal{E}_{\mathtt{G}} \cup \mathcal{E}_{\mathtt{X}}$ with either values or the expression `fail` $\in \mathcal{E}_{\mathtt{G}}$. Finally the rules for scheduling of agents of Figure 10 are such that only expressions in $\mathcal{E}_{\mathtt{G}} \cup \mathcal{E}_{\mathtt{X}}$ and not containing variables are inserted in the running activities. □

Lemma 8 *Let* $\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ *be a non-completed and non-blocked agent instance, let* $\Sigma \Vdash \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R}\rangle^{\mathtt{G}}$ OK *for some* Σ*, and* $\vdash$ `agent G {` `Sns` $\overline{\mathtt{s}}$`;` $\overline{\mathtt{Act}}$`}` OK*. Then*

1. *there is* $\mathcal{R}$*, and* `rdx` *such that* $\mathtt{R} = \mathcal{R}[\![\mathtt{rdx}]\!]$*,* $\mathtt{rdx} \in \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$*. Moreover,*
   (a) *if* `rdx` *is* `started(a)`*, or* `failed(a)`*, or* `completed(a)`*, then* $\mathtt{R} = \mathcal{S}([\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{rdx}]\!]\ \ \overline{\mathtt{Sr}'}]\{\mathtt{e}\}])$*, for some* $\mathcal{S}$*,* $\mathcal{P}$*,* $\mathtt{a}'$*,* $\overline{\mathtt{v}}$*,* $\overline{\mathtt{Sr}}$*,* $\overline{\mathtt{Sr}'}$*, and* e*;*
   (b) *if* `rdx` $=$ `fail` *then* $\mathtt{R} = \mathcal{S}([\mathtt{a}'(\overline{\mathtt{v}})[\ ]\{\mathcal{E}[\![\mathtt{rdx}]\!]\}])$*, for some* $\mathcal{S}$*,*$\mathcal{E}$*,*$\mathtt{a}'$*,* $\overline{\mathtt{v}}$*;*

   or

2. $\mathtt{R} = \mathcal{S}([\mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathtt{a}'(\overline{\mathtt{v}})\langle \mathtt{v}, \mathtt{e}\rangle\ \ \overline{\mathtt{Sr}'}]\{\mathtt{e}'\}])$ *for some for some* $\mathcal{S}$*,* a*,* $\mathtt{a}'$*,* $\overline{\mathtt{w}}$*,* $\overline{\mathtt{v}}$*,* $\overline{\mathtt{Sr}}$*,* $\overline{\mathtt{Sr}'}$*,* v*,* e*, and* $\mathtt{e}'$ *such that:* $\mathtt{v} = \mathtt{tr}$ *implies* $\mathtt{e} = \mathtt{v}'$ *for some* $\mathtt{v}'$
   or

3. $\mathtt{R} = \mathcal{S}([\mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathtt{a}'(\overline{\mathtt{v}})[\,]\{\mathtt{v}\}\ \ \overline{\mathtt{Sr}'}]\{\mathtt{e}\}])$ *for some* $\mathcal{S}$*,* a*,* $\mathtt{a}'$*,* $\overline{\mathtt{w}}$*,* $\overline{\mathtt{v}}$*,* $\overline{\mathtt{Sr}}$*,* $\overline{\mathtt{Sr}'}$*,* v*, and* e*.*

Proof: Since the agent is non-completed $\mathtt{R}$ is $\mathtt{main}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$, and $n = 0$ implies that $\mathtt{e}$ is not a value.
If $n = 0$, from Lemma 1 there is a unique evaluation context $\mathcal{E}$ and redex $\mathtt{rdx}$ such that such that $\mathtt{e} = \mathcal{E}[\![\mathtt{rdx}]\!]$. From Lemma 7 $\mathtt{rdx} \in \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$. Therefore, the evaluation context $\mathcal{R} = \mathtt{main}(\overline{\mathtt{v}})[\,]\{\mathcal{E}\}$ is such that $\mathtt{R} = \mathcal{R}[\![\mathtt{rdx}]\!]$. Moreover, since $\mathtt{e}$ is the body of the $\mathtt{main}$ activity, from $\vdash \mathtt{agent\ G\ \{\ Sns\ \overline{s};\ \overline{Act}\}}$ OK we have that $\mathtt{rdx}$ cannot be $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$. So, 1. holds.
By structural induction on non-blocked activities with some (non zero) subactivities.
Let $\mathtt{R}' = \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$ with $n > 0$. Since $\mathtt{R}'$ is non-blocked there is i, $1 \leq i \leq n$ such that:

(α) $\mathtt{Sr}_i = \mathtt{a}'(\overline{\mathtt{e}})\langle \mathtt{e}', \mathtt{e}'' \rangle$ for some $\mathtt{a}'$, $\overline{\mathtt{e}}$, $\mathtt{e}'$, $\mathtt{e}''$, or

(β) $\mathtt{Sr}_i = \mathtt{a}'(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}']\{\mathtt{e}'\}$ for some $\mathtt{a}'$, $\overline{\mathtt{w}}$, $\mathtt{e}'$, $\overline{\mathtt{Sr}}'$ and $\mathtt{Sr}_i$ is not blocked.

Consider case (α).
If one of the expression in $\overline{\mathtt{e}} = \mathtt{e}_1 \cdots \mathtt{e}_m$ is not a value, say $\mathtt{e}_j$, from Lemma 1 there is a unique evaluation context $\mathcal{E}$ and redex $\mathtt{rdx}$ such that $\mathtt{e}_j = \mathcal{E}[\![\mathtt{rdx}]\!]$. If $\mathtt{rdx}$ is not $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$, let $\mathcal{R}$ be

$$\mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1}\ \ \mathtt{a}'(\overline{\mathtt{v}}\,\mathcal{E}\,\mathtt{e}_{j+1} \cdots \mathtt{e}_m)\langle \mathtt{e}', \mathtt{e}'' \rangle\ \ \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$$

we have that $\mathtt{R}' = \mathcal{R}[\![\mathtt{rdx}]\!]$, and 1. holds.
If $\mathtt{rdx}$ is $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$, let $\mathcal{P} = \mathtt{a}'(\overline{\mathtt{v}}\,\mathcal{E}\,\mathtt{e}_{j+1} \cdots \mathtt{e}_m)\langle \mathtt{e}', \mathtt{e}'' \rangle$ and $\mathcal{S} = (\![\ ]\!)$, we have that $\mathtt{R}' = \mathcal{S}([\![\mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1}\ \ \mathcal{P}[\![\mathtt{rdx}]\!]\ \ \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}]\!])$ and 1(a) holds. Note that $\mathtt{rdx}$ cannot be $\mathtt{fail}$ since by the typing rule for activities in Figure 5, parameters, persistency, and preconditions of subactivities must be side effect free, and therefore cannot contain $\mathtt{fail}$.
Similar if the expressions in $\overline{\mathtt{e}}$ are values and $\mathtt{e}'$ is not a value, or $\overline{\mathtt{e}}$ are values, $\mathtt{e}' = \mathtt{tr}$, and $\mathtt{e}''$ is not a value. (By using the suitable evaluation context.)
If the expressions in $\overline{\mathtt{e}}$ are values and $\mathtt{e}' \neq \mathtt{tr}$, or $\mathtt{e}' = \mathtt{tr}$, and $\mathtt{e}''$ is a value, with $\mathcal{S} = (\![\ ]\!)$ then 2. holds.
Consider case (β).
If $n = 0$ and $\overline{\mathtt{e}'}$ is a value, with $\mathcal{S} = (\![\ ]\!)$ and 3. holds. If $\mathtt{e}'$ is not a value, as before $\mathtt{e}' = \mathcal{E}[\![\mathtt{rdx}]\!]$. Note that, $\mathtt{rdx}$ cannot be $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$ since by the typing rule for activities in Figure 5, the body of an activity may not contain such expressions. If $\mathtt{rdx} \neq \mathtt{fail}$ define $\mathcal{R}$ to be

$$\mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1}\ \ \mathtt{a}'(\overline{\mathtt{w}})[\,]\{\mathcal{E}\}\ \ \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$$

and we have that $\mathtt{R}' = \mathcal{R}[\![\mathtt{rdx}]\!]$, so 1. holds.
In case $\mathtt{rdx} = \mathtt{fail}$ define $\mathcal{S} = \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1} \quad (\![\ ]\!) \ \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$.
Therefore, $\mathtt{R}' = \mathcal{S}(\![\mathtt{a}'(\overline{\mathtt{w}})[\,]\{\mathcal{E}[\![\mathtt{rdx}]\!]\}]\!)$ and 1(b) holds.
If $n > 0$ we can apply the inductive hypothesis to $\mathtt{Sr}_i = \mathtt{a}'(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}']\{\mathtt{e}'\}$ and we have that:

1′. there is $\mathcal{R}$, and $\mathtt{rdx}$ such that $\mathtt{Sr}_i = \mathcal{R}[\![\mathtt{rdx}]\!]$, $\mathtt{rdx} \in \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$. Moreover,

   (a) if $\mathtt{rdx}$ is $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$, then $\mathtt{Sr}_i$ is

$$\mathcal{S}(\![\mathtt{a}^1(\overline{\mathtt{v}}^1)[\overline{\mathtt{Sr}}^1 \quad \mathcal{P}[\![\mathtt{rdx}]\!] \quad \overline{\mathtt{Sr}}^2]\{\mathtt{e}^1\}]\!)$$

   for some $\mathcal{S}$, $\mathcal{P}$, $\mathtt{a}^1$, $\overline{\mathtt{v}}^1$, $\overline{\mathtt{Sr}}^1$, $\overline{\mathtt{Sr}}^2$, and $\mathtt{e}^1$;

   (b) if $\mathtt{rdx} = \mathtt{fail}$ then $\mathtt{Sr}_i$ is $\mathcal{S}(\![\mathtt{a}^1(\overline{\mathtt{v}}^1)[\ ]\{\mathcal{E}[\![\mathtt{rdx}]\!]\}]\!)$, for some $\mathcal{S}$, $\mathcal{E}$, $\mathtt{a}^1$, $\overline{\mathtt{v}}^1$;

   or

2′. $\mathtt{Sr}_i = \mathcal{S}(\![\mathtt{a}^1(\overline{\mathtt{w}}^1)[\overline{\mathtt{Sr}}^1 \quad \mathtt{a}^2(\overline{\mathtt{v}}^1)\langle \mathtt{v}, \mathtt{e}^1\rangle \quad \overline{\mathtt{Sr}}^2]\{\mathtt{e}^2\}]\!)$ for some for some $\mathcal{S}$, $\mathtt{a}^1$, $\mathtt{a}^2$, $\overline{\mathtt{w}}^1$, $\overline{\mathtt{v}}^1$, $\overline{\mathtt{Sr}}^1$, $\overline{\mathtt{Sr}}^2$, $\mathtt{v}^1$, $\mathtt{e}^1$, and $\mathtt{e}^2$ such that: $\mathtt{v}^1 = \mathtt{tr}$ implies $\mathtt{e}^1 = \mathtt{v}^2$ for some $\mathtt{v}^2$
   or

3′. $\mathtt{Sr}_i = \mathcal{S}(\![\mathtt{a}^1(\overline{\mathtt{w}}^1)[\overline{\mathtt{Sr}}^1 \quad \mathtt{a}^2(\overline{\mathtt{v}}^1)[\,]\{\mathtt{v}^1\} \quad \overline{\mathtt{Sr}}^2]\{\mathtt{e}^1\}]\!)$ for some $\mathcal{S}$, $\mathtt{a}^1$, $\mathtt{a}^2$, $\overline{\mathtt{w}}^1$, $\overline{\mathtt{v}}^1$, $\overline{\mathtt{Sr}}^1$, $\overline{\mathtt{Sr}}^2$, $\mathtt{v}^1$, and $\mathtt{e}^1$.

Consider case 1′.
If $\mathtt{rdx}$ is not $\mathtt{fail}$, or $\mathtt{started}(\mathtt{a})$, or $\mathtt{failed}(\mathtt{a})$, or $\mathtt{completed}(\mathtt{a})$, define

$$\mathcal{R}' = \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1} \quad \mathcal{R} \quad \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$$

then $\mathtt{R}' = \mathcal{R}'[\![\mathtt{rdx}]\!]$ and 1. holds. If instead $\mathtt{rdx}$ is either one of the four previously mentioned redexes, define $\mathcal{S}' = \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1} \quad \mathcal{S} \quad \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$, and again 1. holds.
For cases 2′ and 3′ define $\mathcal{S}' = \mathtt{a}(\overline{\mathtt{v}})[\mathtt{Sr}_1 \cdots \mathtt{Sr}_{i-1} \quad \mathcal{S} \quad \mathtt{Sr}_{i+1} \cdots \mathtt{Sr}_n]\{\mathtt{e}\}$ and the corresponding results hold.
□

Lemma 9 (Canonical Lemma) *Let* $\Sigma = \mathtt{typeEnv}(\mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n)$, *and* $\Sigma \Vdash \mathtt{v} : \mathtt{T}$ IN $\mathtt{G}$.

1. *if* $\mathtt{T} = \mathtt{Sns}$, *then* $\mathtt{v} = \sigma$, *and and for some,* k, $1 \leq k \leq n$, $\mathtt{I}_k$ *is* $\sigma = \langle \cdots \rangle^{\mathtt{Sns}}$;

2. *if* $\mathtt{T} = \mathtt{A}$*, then* $\mathtt{v} = \alpha$*, and and for some,* k*,* $1 \leq k \leq n$*,* $\mathtt{I}_k$ *is* $\alpha = \langle \cdots \rangle^{\mathtt{A}}$*;*

3. *if* $\mathtt{T} = \mathtt{Bool}$*, then* $\mathtt{v} = \mathtt{tr}$*, or* $\mathtt{v} = \mathtt{fls}$*;*

4. *if* $\mathtt{T} = \mathtt{typeOf(c)}$*, then* $\mathtt{v} = \mathtt{c}$*.*

Proof: By case analysis on the typing rules. □

Lemma 10 (Non-blocked agent progress lemma) *Let* $\mathtt{M} = \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ *be such that* $\vdash \mathtt{M}$ OK*, and let* $\mathtt{I}_k$ *(for some* $k \in 1..n$*) be an agent instance not completed and not blocked in* M*. Then* $\mathtt{M} \Rightarrow \mathtt{M}'$ *for some* $\mathtt{M}'$*.*

Proof: Let $\mathtt{I}_k$ be $\gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R} \rangle^{\mathtt{G}}$. Let $\mathtt{typeEnv(M)} = \Sigma$. From $\vdash \mathtt{M}$ OK we have that $\Sigma \Vdash \gamma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}}, \overline{\sigma}, \mathtt{R} \rangle^{\mathtt{G}}$ OK. Therefore, $\Sigma \Vdash \mathtt{R}$ OK IN G and for all $\sigma \in \overline{\sigma}$ we have that $\Sigma(\sigma) = \mathtt{Sns}$, i.e., there is j, $j \in 1..n$, such that $\mathtt{I}_j$ is $\sigma = \langle \overline{\mathtt{l}}\,\overline{\mathtt{v}} \rangle^{\mathtt{Sns}}$.

Since we are assuming a well-typed program, $\vdash \mathtt{agent\ G\ \{\ Sns\ \bar{s};\ \overline{Act}\}}$ OK, from Lemma 8 we have that

1. there is $\mathcal{R}$, and rdx such that $\mathtt{R} = \mathcal{R}[\![\mathtt{rdx}]\!]$, $\mathtt{rdx} \in \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$. Moreover,
   (a) if rdx is started(a), or failed(a), or completed(a), then $\mathtt{R} = \mathcal{S}([\mathtt{a}'(\overline{\mathtt{v}})[\overline{\mathtt{Sr}}\ \ \mathcal{P}[\![\mathtt{rdx}]\!]\ \ \overline{\mathtt{Sr}}']\{\mathtt{e}\}])$, for some $\mathcal{S}$, $\mathcal{P}$, $\mathtt{a}'$, $\overline{\mathtt{v}}$, $\overline{\mathtt{Sr}}$, $\overline{\mathtt{Sr}}'$, and e;
   (b) if $\mathtt{rdx} = \mathtt{fail}$ then $\mathtt{R} = \mathcal{S}([\mathtt{a}'(\overline{\mathtt{v}})[\ ]\{\mathcal{E}[\![\mathtt{rdx}]\!]\}])$, for some $\mathcal{S}$, $\mathcal{E}$, $\mathtt{a}'$, $\overline{\mathtt{v}}$;

   or

2. $\mathtt{R} = \mathcal{S}([\mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathtt{a}'(\overline{\mathtt{v}})\langle \mathtt{v}, \mathtt{e} \rangle\ \ \overline{\mathtt{Sr}}']\{\mathtt{e}'\}])$ for some for some $\mathcal{S}$, a, $\mathtt{a}'$, $\overline{\mathtt{w}}$, $\overline{\mathtt{v}}$, $\overline{\mathtt{Sr}}$, $\overline{\mathtt{Sr}}'$, v, e, and $\mathtt{e}'$ such that: $\mathtt{v} = \mathtt{tr}$ implies $\mathtt{e} = \mathtt{v}'$ for some $\mathtt{v}'$
   or

3. $\mathtt{R} = \mathcal{S}([\mathtt{a}(\overline{\mathtt{w}})[\overline{\mathtt{Sr}}\ \ \mathtt{a}'(\overline{\mathtt{v}})[\,]\{\mathtt{v}\}\ \ \overline{\mathtt{Sr}}']\{\mathtt{e}\}])$ for some $\mathcal{S}$, a, $\mathtt{a}'$, $\overline{\mathtt{w}}$, $\overline{\mathtt{v}}$, $\overline{\mathtt{Sr}}$, $\overline{\mathtt{Sr}}'$, v, and e.

Consider the three cases.

1. By cases on the redex $\mathtt{rdx} \in \mathcal{X}_{\mathtt{G}} \cup \mathcal{X}_{\mathtt{X}}$.
   For most of the redexes the corresponding reduction rule does not have restrictions. We only analyze the ones that require the well-typedness of the configuration in order to reduce.
   If $\underline{\mathtt{rdx} = \mathtt{spawn\,G}(\overline{\mathtt{v}})}$ from the fact that the program is well-typed, G

is a defined agent and it has a `main` activity, and, from the typing rule [TnewG] the number of actual parameters matches the one of the formal parameters of the `main` activity. Therefore, [AGN] is applicable. Note that, subject reduction ensure that also the types are matching.
If $\underline{\texttt{rdx} = \texttt{make}\,\texttt{A}(\overline{\texttt{v}})}$, from the fact that the program is well-typed, `A` is a defined artifact, and, from typing rule [TnewA] the number of actual parameters matches the number of fields and properties of the artifact. Therefore, [ART] is applicable.
If $\underline{\texttt{rdx} = \texttt{.s}}$, from typing rule [Tsns], we have that $\texttt{s} \in \bar{\texttt{s}}$, say $\texttt{s} = \texttt{s}_i$, and $\texttt{Sns}\ \bar{\texttt{s}} \in \texttt{G}$. Moroever, from $\gamma = \langle \overline{\texttt{l}}\,\overline{\texttt{v}}, \overline{\sigma}, \texttt{R}\rangle^{\texttt{G}}$ the ith sensor instance exists. Therefore, reduction rule [SNS] is applicable.
If $\texttt{rdx} = \texttt{sense}\ \texttt{v}\ \texttt{:filter}\ \texttt{l}$, from typing rule [Tperc], $\Sigma \Vdash \texttt{v} : \texttt{Sns}$ IN G. Therefore, from Lemma 9, $\texttt{v} = \sigma$, and for some, k, $1 \leq k \leq n$, $\texttt{I}_k$ is $\sigma = \langle \overline{\texttt{l}}\,\overline{\texttt{v}}\rangle^{\texttt{Sns}}$. If $\texttt{l} = \texttt{l}_i$ for some i, then rule [PER] is applicable. If not, since the agent is not blocked, either there are other expressions reducible or case 2. or 3. hold, and therefore the term would reduce.
If $\underline{\texttt{rdx} = \texttt{use}\ \texttt{v}_2.\texttt{o}(\overline{\texttt{v}})\ \texttt{:sns}\ \texttt{v}_1}$, from typing rule [Top], $\Sigma \Vdash \texttt{v}_2 : \texttt{A}$ IN G, and the operation `o` is defined in `A` and as as many formal parameters as the values in $\overline{\texttt{v}}$. From Lemma 9, $\texttt{v}_2 = \alpha$, and for some, k, $1 \leq k \leq n$, $\texttt{I}_k$ is $\alpha = \langle \cdots \rangle^{\texttt{A}}$. Therefore, reduction rule [USE] is applicable.
If $\underline{\texttt{rdx} = \texttt{focus}(\texttt{v}_1, \texttt{v}_2)}$ from typing rule [Tfocus], $\Sigma \Vdash \texttt{v}_2 : \texttt{A}$ IN G. From Lemma 9, $\texttt{v}_2 = \alpha$, and for some, k, $1 \leq k \leq n$, $\texttt{I}_k$ is $\alpha = \langle \cdots \rangle^{\texttt{A}}$. Therefore, reduction rule [FOC] is applicable. Similar for $\texttt{unfocus}(\texttt{v}_1, \texttt{v}_2)$.
If $\underline{\texttt{rdx} = \texttt{observe}\ \texttt{v}_1.\texttt{p}}$ from typing rule [TpropA], $\Sigma \Vdash \texttt{v}_1 : \texttt{A}$ IN G, and `p` is a property defined in `A`. From Lemma 9, $\texttt{v}_1 = \alpha$, and for some, k, $1 \leq k \leq n$, $\texttt{I}_k$ is $\alpha = \langle \cdots \rangle^{\texttt{A}}$. Therefore, reduction rule [GETA] is applicable.

2. From $\Sigma \Vdash$ R OK IN G we have $\Sigma \Vdash \texttt{a}'(\overline{\texttt{v}})\langle \texttt{v}, \texttt{e}\rangle$ OK IN G. Therefore, $\Sigma \Vdash \texttt{v} : \texttt{Bool}$ IN G, and $\Sigma \Vdash \texttt{e} : \texttt{Bool}$ IN G. From Lemma 9, $\texttt{v} = \texttt{fls}$ or $\texttt{v} = \texttt{tr}$. If $\texttt{v} = \texttt{fls}$ then rule [DISP] is applicable. If $\texttt{v} = \texttt{tr}$, then $\texttt{e} = \texttt{v}'$ for some $\texttt{v}'$. From Lemma 9 then $\texttt{e} = \texttt{fls}$ or $\texttt{e} = \texttt{tr}$. In the first case rule [PREC] is applicable and in the second rule [SCH].

3. In this case rule [END-SA] is applicable.

□

Lemma 11 *Let* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m \rangle^{\mathtt{A}}$ *be an artifact instance, and let* $\Sigma \Vdash \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m \rangle^{\mathtt{A}}$ OK *for some* Σ*, and* $\vdash$ `artifact A{`$\overline{\mathtt{U}}\ \overline{\mathtt{f}}$`;` $\overline{\mathtt{V}}\ \overline{\mathtt{p}}$`;` $\overline{\mathtt{Op}}\ \overline{\mathtt{Step}}$`}` OK*. Then, for all* i*,* $1 \leq i \leq m$ *we have that* $\mathtt{O}_i$ *contains only (sub)expressions in* $\mathcal{E}_{\mathtt{A}} \cup \mathcal{E}_{\mathtt{X}}$*, identifiers in* $\overline{\mathtt{f}}$ *or* $\overline{\mathtt{p}}$*, i.e.,* R *does not contain variables, or (sensor) identifiers, or expressions in* $\mathcal{E}_{\mathtt{G}}$*;*

Proof: From $\Sigma \Vdash \alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m \rangle^{\mathtt{A}}$ OK we have that, for all i, $1 \leq i \leq m$, $\Sigma \Vdash \mathtt{O}_i$ OK IN A. Observe that the typing rules of Figure 4 and 6 are such that only expressions in $\mathcal{E}_{\mathtt{A}} \cup \mathcal{E}_{\mathtt{X}}$ are allowed in operations or steps. The rule for artifact creation, [ART] of Figure 8, does not generate any expression, and the reduction rules of Figure 11 replace expressions in $\mathcal{E}_{\mathtt{A}} \cup \mathcal{E}_{\mathtt{X}}$ with values. Finally the rules for state change of artifacts of Figure 12 are such that only expressions in $\mathcal{E}_{\mathtt{A}} \cup \mathcal{E}_{\mathtt{X}}$ and not containing variables are inserted in the artifact instance. □

Lemma 12 *Let* I *be* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m \rangle^{\mathtt{A}}$*, a non-idle artifact instance, i.e.* $m > 0$*, and let* $\Sigma \Vdash$ I OK *for some* Σ*, and* $\vdash$ `artifact A{`$\overline{\mathtt{U}}\ \overline{\mathtt{f}}$`;` $\overline{\mathtt{V}}\ \overline{\mathtt{p}}$`;` $\overline{\mathtt{Op}}\ \overline{\mathtt{Step}}$`}` OK*. Then*

1. *there is* $\mathcal{O}$*, and* $\mathtt{rdx} \in \mathcal{X}_{\mathtt{X}} \cup \mathcal{X}_{\mathtt{A}}$ *such that* I *is* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ \mathcal{O}[\![\mathtt{rdx}]\!] \rangle^{\mathtt{A}}$*. Moreover, if* $\mathtt{rdx} \neq \mathtt{.f}$*, and* $\mathtt{rdx} \neq \mathtt{.p}$*,* I *is*

$$\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr} \rangle \{\mathcal{E}[\![\mathtt{rdx}]\!]\}) \rangle^{\mathtt{A}}$$

   for some $\overline{\mathtt{v}}$*,* σ*,* o*,* $\overline{\mathtt{St}}$*,* $\mathcal{E}$*, or*

2. I *is* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_n]\langle \mathtt{tr} \rangle \{\mathtt{v}\}) \rangle^{\mathtt{A}}$*,* $n \geq 0$*, for some* $\overline{\mathtt{v}}$*,* σ*,* o*,* $\overline{\mathtt{St}}$*, or*

3. I *is* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{fls} \rangle \{\mathtt{e}\}) \rangle^{\mathtt{A}}$ *for some* $\overline{\mathtt{v}}$*,* σ*,* o*,* e*,* $\mathcal{E}$*, or*

4. I *is* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}[\overline{\mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{fls} \rangle}\ \ \mathtt{o}'(\overline{\mathtt{v}}')\langle \mathtt{tr} \rangle\ \ \overline{\mathtt{St}}]) \rangle^{\mathtt{A}}$ *for some* $\overline{\mathtt{v}}'$*,* σ*,* o*,* o′*,* $\mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{fls} \rangle$*,* o*,* $\overline{\mathtt{St}}$*, or*

5. I *is* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}[\mathtt{o}_1(\overline{\mathtt{v}}^1)\langle \mathtt{fls} \rangle \cdots \mathtt{o}_n(\overline{\mathtt{v}}^n)\langle \mathtt{fls} \rangle]) \rangle^{\mathtt{A}}$ *for some* $\overline{\mathtt{v}}^i$*,* $\mathtt{o}_i$*,*$(1 \leq i \leq n)$*,* σ*,* o*.*

Proof: Let I be $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m \rangle^{\mathtt{A}}$, a non-idle artifact instance. Then I is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ \mathtt{O} \rangle^{\mathtt{A}}$, where O is

(a) $(\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathtt{e}_1 \rangle \{\mathtt{e}_2\})$, or

(b) $(\sigma, \mathtt{o}[\mathtt{o}_1(\overline{\mathtt{v}}^1)\langle \mathtt{e}_1\rangle \cdots \mathtt{o}_q(\overline{\mathtt{v}}^q)\langle \mathtt{e}_q\rangle])$, $p \geq 0$, and $q \geq 1$

Consider case (a).
Assume first that $\mathtt{e}_1 = \mathtt{v}$ for some $\mathtt{v}$. Since $\Sigma \Vdash$ `I` OK, from the typing rule for operations of Figure 14 we have that $\Sigma \Vdash \mathtt{v} : \mathtt{Bool}$ IN `A`. From Lemma 9, then $\mathtt{v} = \mathtt{tr}$ or $\mathtt{v} = \mathtt{fls}$. If $\mathtt{v} = \mathtt{fls}$, then 3. holds. If $\mathtt{v} = \mathtt{tr}$ and $\mathtt{e}_2 = \mathtt{v}'$ for some $\mathtt{v}'$. Then 2. holds. Otherwise, from Lemma 1 there is a (unique) expression evaluation context, $\mathcal{E}$, and redex, `rdx`, such that $\mathtt{e}_2 = \mathcal{E}[\![\mathtt{rdx}]\!]$. From Lemma 11 $\mathtt{rdx} \in \mathcal{X}_\mathtt{X} \cup \mathcal{X}_\mathtt{A}$. Define $\mathcal{O}$ to be $(\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathtt{tr}\rangle\{\mathcal{E}\})$, 1. holds.
Assume that $\mathtt{e}_1$ is not a value. From Lemma 1 there is a (unique) expression evaluation context, $\mathcal{E}$, and redex, `rdx`, such that $\mathtt{e}_1 = \mathcal{E}[\![\mathtt{rdx}]\!]$. From Lemma 11 $\mathtt{rdx} \in \mathcal{X}_\mathtt{X} \cup \mathcal{X}_\mathtt{A}$. Define $\mathcal{O}$ to be $(\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_p]\langle \mathcal{E}\rangle\{\mathtt{e}_2\})$. We have that $\mathtt{O}\mathcal{O}[\![\mathtt{rdx}]\!]$. From $\Sigma \Vdash$ `I` OK we also have that `rdx` is side effect free and therefore may only be either `.f`, or `.p`. Therefore, 1. holds.
Consider case (b).
Let assume first $(\sigma, \mathtt{o}[\mathtt{o}_1(\overline{\mathtt{v}}^1)\langle \mathtt{v}_1\rangle \cdots \mathtt{o}_q(\overline{\mathtt{v}}^q)\langle \mathtt{v}_q\rangle])$. First, note that, since $\Sigma \Vdash$ `I` OK, from the typing rule for steps of Figure 14 we have that $\Sigma \Vdash \mathtt{v}_i : \mathtt{Bool}$ IN `A`, $1 \leq i \leq q$. From Lemma 9, then $\mathtt{v}_i = \mathtt{tr}$ or $\mathtt{v}_i = \mathtt{fls}$. Therefore, either 4. or 5. holds (depending on the facts that the guards are all `fls` or there is a `tr` and so a first one).
Assume that $(\sigma, \mathtt{o}[\mathtt{o}_1(\overline{\mathtt{v}}^1)\langle \mathtt{v}_1\rangle \cdots \mathtt{o}_j(\overline{\mathtt{v}}^j)\langle \mathtt{e}_j\rangle \cdots \mathtt{o}_q(\overline{\mathtt{v}}^q)\langle \mathtt{e}_q\rangle])$, and $\mathtt{e}_j$ is the first guard which is not a value. From Lemma 1 there is a (unique) expression evaluation context, $\mathcal{E}$, and redex, `rdx`, such that $\mathtt{e}_j = \mathcal{E}[\![\mathtt{rdx}]\!]$. From the previous point we can assume that all the $\mathtt{v}_k$, $1 \leq k \leq j-1$ are such that $\mathtt{v}_k = \mathtt{fls}$. Define the evaluation context $\mathcal{O}$ to be $(\sigma, \mathtt{o}[\overline{\mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{fls}\rangle}\ \ \mathtt{o}_j(\overline{\mathtt{v}}^j)\langle \mathcal{E}\rangle \cdots \mathtt{o}_q(\overline{\mathtt{v}}^q)\langle \mathtt{e}_q\rangle])$. From the typing rule for steps of Figure 14 we have that `rdx` is side effect free and therefore may only be either `.f`, or `.p`. Therefore, 1. holds. □

Lemma 13 (Non-idle artifact progress lemma) *Let* $\vdash \mathtt{M}$ OK, $\mathtt{M} = \mathtt{I}_1 \mid \cdots \mid \mathtt{I}_n$ *and (for some* $k \in 1..n$*)* $\mathtt{I}_k$ *is a non-idle artifact* $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m\rangle^\mathtt{A}$. *Then* $\mathtt{M} \Rightarrow \mathtt{M}'$ *for some* $\mathtt{M}'$.

Proof: Let $\mathtt{I}_k$ be $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \mathtt{O}_1 \cdots \mathtt{O}_m\rangle^\mathtt{A}$. Let $\mathtt{typeEnv}(\mathtt{M}) = \Sigma$. From $\vdash \mathtt{M}$ OK we have that $\Sigma \Vdash \mathtt{I}_k$ OK. Since we are assuming a well-typed program, $\vdash \mathtt{artifact\ A}\{\overline{\mathtt{U}}\ \overline{\mathtt{f}};\ \overline{\mathtt{V}}\ \overline{\mathtt{p}};\ \overline{\mathtt{Op}}\ \overline{\mathtt{Step}}\}$ OK. From Lemma 11 we have that

1. there is $\mathcal{O}$, and $\mathtt{rdx} \in \mathcal{X}_\mathtt{X} \cup \mathcal{X}_\mathtt{A}$ such that `I` is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} =$

$\overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ \mathcal{O}[\![\mathtt{rdx}]\!]\rangle^{\mathtt{A}}$. Moreover, if $\mathtt{rdx} \neq .\mathtt{f}$, and $\mathtt{rdx} \neq .\mathtt{p}$, $\mathtt{I}$ is

$$\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr}\rangle\{\mathcal{E}[\![\mathtt{rdx}]\!]\})\rangle^{\mathtt{A}}$$

for some $\overline{\mathtt{v}}$, σ, $\mathtt{o}$, $\overline{\mathtt{St}}$, $\mathcal{E}$, or

2. $\mathtt{I}$ is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\mathtt{St}_1 \cdots \mathtt{St}_n]\langle \mathtt{tr}\rangle\{\mathtt{v}\})\rangle^{\mathtt{A}}$, $n \geq 0$, for some $\overline{\mathtt{v}}$, σ, $\mathtt{o}$, $\overline{\mathtt{St}}$, or

3. $\mathtt{I}$ is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{fls}\rangle\{\mathtt{e}\})\rangle^{\mathtt{A}}$ for some $\overline{\mathtt{v}}$, σ, $\mathtt{o}$, $\mathtt{e}$, $\mathcal{E}$, or

4. $\mathtt{I}$ is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}[\overline{\mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{fls}\rangle}\ \ \mathtt{o}'(\overline{\mathtt{v}}')\langle \mathtt{tr}\rangle\ \ \overline{\mathtt{St}}])\rangle^{\mathtt{A}}$ for some $\overline{\mathtt{v}}'$, σ, $\mathtt{o}$, $\mathtt{o}'$, $\overline{\mathtt{o}(\overline{\mathtt{v}})\langle \mathtt{fls}\rangle}$, $\mathtt{o}$, $\overline{\mathtt{St}}$, or

5. $\mathtt{I}$ is $\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}[\mathtt{o}_1(\overline{\mathtt{v}}^1)\langle \mathtt{fls}\rangle \cdots \mathtt{o}_n(\overline{\mathtt{v}}^n)\langle \mathtt{fls}\rangle])\rangle^{\mathtt{A}}$ for some $\overline{\mathtt{v}}^i$, $\mathtt{o}_i$,$(1 \leq i \leq n)$, σ, $\mathtt{o}$.

Consider the five cases.

1. By cases on the redex $\mathtt{rdx} \in \mathcal{X}_{\mathtt{A}} \cup \mathcal{X}_{\mathtt{X}}$.
   If $\mathtt{rdx} = \mathtt{spawn}\,\mathtt{G}(\overline{\mathtt{v}})$ or $\mathtt{rdx} = \mathtt{make}\,\mathtt{A}(\overline{\mathtt{v}})$, is like the corresponding cases of Lemma 10.
   If $\mathtt{rdx} = .\mathtt{f}$, from the typing rule [TfieldR] of Figure 4 we have that $\mathtt{f} \in \overline{\mathtt{f}}$. Therefore, rule [GET] is applicable.
   If $\mathtt{rdx} = .\mathtt{p}$, from the typing rule [TpropR] of Figure 4 we have that $\mathtt{p} \in \overline{\mathtt{p}}$. Therefore, rule [GETPR] is applicable.
   For all the other redexes,

   $$\alpha = \langle \overline{\mathtt{f}} = \overline{\mathtt{v}}, \overline{\mathtt{p}} = \overline{\mathtt{w}}, \overline{\sigma}, \overline{\mathtt{O}}\ \ (\sigma, \mathtt{o}(\overline{\mathtt{v}})[\overline{\mathtt{St}}]\langle \mathtt{tr}\rangle\{\mathcal{E}[\![\mathtt{rdx}]\!]\})\rangle^{\mathtt{A}}$$

   for some $\overline{\mathtt{v}}$, σ, $\mathtt{o}$, $\overline{\mathtt{St}}$, $\mathcal{E}$. If $\mathtt{rdx} = .\mathtt{f} = \mathtt{v}$, and $\mathtt{rdx} = .\mathtt{p} = \mathtt{v}$, similar to the [GET] and [GETPR] cases.
   If $\mathtt{rdx} = \mathtt{signal}(\mathtt{l}(\mathtt{v}))$, since the configuration is well-typed, from the rules in Figure 14 the sensors in $\overline{\sigma}$ and σ are defined, therefore [GEN]is applicable.
   If $\mathtt{rdx} = \mathtt{next}\ \mathtt{o}(\overline{\mathtt{v}})$, from the typing rule [Tnext] of Figure 4 we have that $\mathtt{step}\ \mathtt{o}(\overline{\mathtt{U}}\ \overline{\mathtt{x}}) \cdots \in \mathtt{A}$, therefore rule [NEXT] is applicable.

2. If $n > 0$ rule [SELG] is applicable, and for $n = 0$ [END] is applicable.

3. Rule [FLS] is applicable.

4. Rule [SGO] is applicable.
5. Rule [SKIP] is applicable.

□

Proof of Theorem 2 (Progress):

By Lemmas 10 and 13.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] Gul Agha, Peter Wegner, and Akinori Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, Cambridge, MA, USA, 1993.

[4] Joe Armstrong. Erlang. *Commun. ACM*, 53:68–75, September 2010. doi:10.1145/1810891.1810910.

[5] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[6] Nick Benton, Luca Cardelli, and Cedric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004. doi:10.1145/1018203.1018205.

[7] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 2)*. Springer, 2009.

[8] Rafael Bordini, Mendi Dastani, Jurgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 1)*. Springer, 2005.

[9] Rafael Bordini, Jomi Hübner, and Mike Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[10] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998. doi:10.1145/292469.292470.

[11] Tony Clark. Specification and implementation of a multi-agent calculus based on higher-order functions. Technical report, Department of Computing, University of Bradford, 1999.

[12] J-L. Colaco, M. Pantel, and P. Sall. CAP: An actor dedicated process calculus. *In the Proceedings of ECOOP96 Workshop on Proof Theory of Concurrent Object-Oriented Programming (PTCOOP96)*, 1996.

[13] Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. On Re-classification and Multi-threading. *Journal of Object Technology (www.jot.fm)*, 3(11):5–30, 2004. doi:10.5381/jot.2004.3.11.a1.

[14] Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Re-classification and multi-threading: FickleMT. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 1297–1304, New York, NY, USA, 2004. ACM. doi:10.1145/967900.968163.

[15] Ferruccio Damiani, Elena Giachino, Paola Giannini, and Emanuele Cazzola. On state classes and their dynamic semantics. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *Software and Data Technologies*, volume 10 of *Communications in Computer and Information Science*, pages 84–96. Springer, 2008. doi:10.1007/978-3-540-70621-2_8.

[16] Ferruccio Damiani, Elena Giachino, Paola Giannini, and Sophia Drossopoulou. A type safe state abstraction for coordination in Java-like languages. *Acta Informatica*, 45(7-8):479–536, 2008. doi:10.1007/s00236-008-0079-y.

[17] Ferruccio Damiani, Paola Giannini, Alessandro Ricci, and Mirko Viroli. FEATHERWEIGHT AGENT LANGUAGE - A Core Calculus for Agents and Artifacts. In *ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies*, volume 1, pages 218–225. INSTICC Press, 2009.

[18] Ferruccio Damiani, Paola Giannini, Alessandro Ricci, and Mirko Viroli. A calculus of agents and artifacts. In Jose Cordeiro, AlpeshKumar Ranchordas, and Boris Shishkov, editors, *Software and Data Technologies*,

volume 50 of *Communications in Computer and Information Science*, pages 124–136. Springer, 2011. doi:10.1007/978-3-642-20116-5_10.

[19] Mehdi Dastani. 2APL: a Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008. doi:10.1007/s10458-008-9036-y.

[20] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998. doi:10.1109/32.685256.

[21] Louise A. Dennis, Berndt Farwer, Rafael H. Bordini, Michael Fisher, and Michael Wooldridge. A common semantic basis for BDI languages. In Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2008. doi:10.1007/978-3-540-79043-3_8.

[22] Rogier M. van Eijk, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Operational semantics for agent communication languages. In *Issues in Agent Communication*, pages 80–95, London, UK, UK, 2000. Springer. doi:10.1007/10722777_6.

[23] Jacques Ferber and Olivier Gutknecht. Operational semantics of multi-agent organizations. In *Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL)*, volume 1757 of *Lecture Notes in Computer Science*, pages 205–217. Springer, 2000. doi:10.1007/10719619_15.

[24] Cédric Fournet and Georges Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332, London, UK, 2002. Springer. doi:10.1007/3-540-45699-6_6.

[25] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 406–421, London, UK, 1996. Springer. doi:10.1007/3-540-61604-7_67.

[26] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th international conference on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-72794-1_10.

[27] Matthew Hennessey and James Riely. Type-safe execution of mobile agents in anonymous networks. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet programming*, pages 95–115. Springer, London, UK, UK, 1999. doi:10.1007/3-540-48749-2_5.

[28] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Inf. Comput.*, 173(1):82–120, February 2002. doi:10.1006/inco.2001.3089.

[29] Koen Hindriks, Frank de Boer, Wiebe van der Hoek, and John-Jules Meyer. Formal semantics for an abstract agent programming language. In Munindar Singh, Anand Rao, and Michael Wooldridge, editors, *Intelligent Agents IV Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 1998. doi:10.1007/BFb0026761.

[30] Koen V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications (2nd volume)*, pages 3–37. Springer, 2009.

[31] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991. doi:10.1007/BFb0057019.

[32] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[33] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328472.

[34] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

[35] Atsushi Igarashi. A Featherweight Approach to FOOL. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, page 433. Springer, 2011. doi:10.1007/978-3-642-22655-7.

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

[37] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001. doi:10.1145/367211.367250.

[38] Christoph G. Jung and Klaus Fischer. A layered agent calculus with concurrent, continuous processes. In *Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, pages 245–258, London, UK, 1998. Springer. doi:10.1007/BFb0026763.

[39] Rajesh K. Karmani and Gul Agha. Actors. In *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011.

[40] Rajesh Kumar Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A Comparative Analysis. *In the proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, pages 11–20, 2009. doi:10.1145/1596655.1596658.

[41] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes*

in Computer Science, pages 439–453. Springer, 2003. doi:10.1007/978-3-540-40007-3_26.

[42] Open System Laboratory. ActorFoundry, http://osl.cs.uiuc.edu/af/.

[43] Y. Lesperance, H. J. Levesque, and R. Reiter. A situation calculus approach to modeling and programming agents. In *Foundations of Rational Agency*, pages 275–299. Kluwer, 1999.

[44] T. Massart. An agent calculus with simple actions where the enabling and disabling are derived operators. *Information Processing Letters*, 40(4):213 – 218, 1991. doi:10.1016/0020-0190(91)90080-2.

[45] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. doi:10.1016/0890-5401(92)90008-4.

[46] Alvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication. In Joao Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Computer Science*, pages 135–154. Springer, 2004. doi:10.1007/978-3-540-25932-9_8.

[47] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A Meta-Model for Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 17, 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems. doi:10.1007/s10458-008-9053-x.

[48] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. doi:10.1109/LICS.1993.287570.

[49] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *First International Conference on Multi Agent Systems (ICMAS95)*, 1995.

[50] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on*

Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer.

[51] Alessandro Ricci and Andrea Santi. Typing Multi-Agent Programs in simpAL. In *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS'12)*, pages 132–147, 2012.

[52] Alessandro Ricci, Mirko Viroli, and Maurizio Cimadamore. Prototyping Concurrent Systems with Agents and Artifacts: Framework and Core Calculus. *Electron. Notes Theor. Comput. Sci.*, 194(4):111–132, 2008. doi:10.1016/j.entcs.2008.03.102.

[53] Alessandro Ricci, Mirko Viroli, and Giulio Piancastelli. simpA: An agent-oriented approach for programming concurrent applications on top of Java. *Science of Computer Programming*, 76(1):37 – 62, 2011. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures - FOCLASA'07. doi:10.1016/j.scico.2010.06.012.

[54] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach (second edition)*. Prentice Hall, 2003.

[55] Peter Sewell, Pawel T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *ICCL Workshop: Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1998. doi:10.1007/3-540-47959-7_1.

[56] Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32(4), 2010. doi:10.1145/1734206.1734209.

[57] Alexandru Suna and Amal El Fallah-Seghrouchni. Programming mobile intelligent agents: An operational semantics. *Web Intelligence and Agent Systems*, 5(1):47–67, 2007.

[58] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62, September 2005. doi:10.1145/1095408.1095421.

[59] Danny Weyns, Andrea Omicini, and James J. Odell. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007. Special Issue on Environments for Multi-agent Systems. doi:10.1007/s10458-006-0012-0.

[60] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995. doi:10.1017/S0269888900008122.

[61] Mike Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley & Sons, Ltd, 2002.

[62] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1(115):38–94, 1994. doi:10.1006/inco.1994.1093.

[63] A Yonezawa and M Tokoro, editors. *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, MA, USA, 1986.

[64] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Inf. Comput.*, 174(2):143–179, May 2002. doi:10.1006/inco.2002.3113.