Typing Multi-agent Programs in simpAL

Alessandro Ricci and Andrea Santi

DISI, University of Bologna
via Venezia 52, 47023 Cesena, Italy
{a.ricci,a.santi}@unibo.it

Abstract. Typing is a fundamental mechanism adopted in mainstream programming languages, important in particular when developing programs of a certain complexity to catch errors at compile time, before executing a program, and to improve the overall design of a system. In this paper we introduce typing also in agent-oriented programming, by using a novel agent programming language called simpAL, which has been conceived from scratch to have this feature.

1 Introduction

Typing is an important mechanism introduced in traditional programming languages, particularly useful if not indispensable when developing programs of a certain complexity [19,16,6,4]. Generally speaking, the definition of a (strong and static) type system in a programming language brings two main benefits. First, it enables compile time error checking, greatly reducing the cost of errors detection—from both a temporal and economic point of view. Second, it provides developers with a conceptual tool for modeling generalization/specialization relationships among concepts and abstractions, eventually specializing existing ones through the definition of proper sub-types and making it possible to fully exploit the principle of substitutability [29] for supporting a safe extension and reuse in programming.

We argue that these features could be very useful and important also for agent-oriented programming (AOP), in particular as soon as AOP is investigated as a paradigm for developing software systems in general [25]. To authors' knowledge, there are no agent-oriented programming languages (APLs) in the state-of-the-art that fully support typing and related features. Consequently, the support which is provided by existing languages to catch errors *before* executing the system is quite weak. To this purpose, in this paper we describe an approach that introduces typing in agent-oriented programming, in particular by means of a novel agent programming language called simpAL, which has been conceived from scratch to have this feature. simpAL, whose general design and concepts have been recently introduced elsewhere [26], has been conceived on the one side drawing inspiration from existing APLs based on the BDI model [23] – AgentSpeak(L) [22] / Jason [2] in particular – and existing meta-models such as the A&A [18] (Agents and Artifacts), along with related frameworks such as

M. Dastani, J.F. Hübner, and B. Logan (Eds.): ProMAS 2012, LNAI 7837, pp. 138–157, 2013.

CArtAgO [24]. On the other side, it has been designed having in mind agent-oriented programming as an evolution of Object-Oriented Programming, to be explored as a paradigm for general-purpose computing and software development [25]. Generally speaking, simpAL is not meant to be as flexible and effective as existing APLs for tackling the development of agent-based systems in the context of Distributed Artificial Intelligence, but it is meant to provide more robust and effective features for the development of general software systems yet characterized by elements of complexity related to concurrency, distribution, decentralization of control, reactivity, etc. In that perspective, typing – as well as other mechanisms not considered in the paper such as *inheritance* – is considered an essential feature.

The remainder of the paper is organized as follows: In Section 2, first we briefly remind the role of typing for programming in general, then we discuss what kind of errors we aim at detecting by introducing typing in agent-oriented programming, as an improvement of the current error checking support provided by existing APLs in the state-of-the-art. Section 3 contains the core contribution of the paper, which is about introducing typing in agent-oriented programming, taking simpAL as target programming language. Finally, in Section 4 we discuss related work and in Section 5 we provide concluding remarks.

2 Bringing Types in Agent-Oriented Programming: Desiderata

2.1 On the Role of Typing for Programming

In the context of programming and software development, typing plays an important role in helping programmers organise computational structures and use them correctly [19,17]. In general, a *type* is a collection of computational entities that share some common property. A *type system* can be defined as *a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute* [19]. In the most general case, type systems and type theory refer to a broad field of study in logics, mathematics and philosophy. Here we consider their specific applications in programming languages, where three main uses of types can be identified [17]:

- *Detecting errors* – type errors occur when a computational entity, such as a function or a data value, is used in a manner that is inconsistent with the concept it represents. For instance, in the case of OO programming languages, invoking a method which is not part of the object (class) interface or passing wrong parameters, or rather assigning wrong values to an object's instance fields. Static type checking allows early detection of these kind of errors, that can be fixed then before running the program.
- *Program organisation and documentation* – in modern programming languages types can be used to represent concepts related to the problem to be solved and their relationships, providing an important support for the high-level organisation of programs, improving their readability, understanding

and maintenance. For instance, in a object-oriented CAD program using an interface/type *Shape* with some *draw* method to represent geometrical shapes, and different classes (*Rectangle*, *Circle*, etc.) – one for each specific concrete shape and a concrete draw behavior. Type systems enforce disciplined programming and this is important in particular in the context of large-scale software composition, where they typically form the backbone of the modules used to package and tie together the components of a large system. Module's interfaces are typically seen as the types of the module.

– *Efficiency* – typing makes it possible to avoid (some) error checking at runtime (since it has been done already at compile time), so improving performance. Generally speaking, types provide information to the compiler about the computational entities in the program that could be useful to produce optimized code to be executed.

These uses are not bound to any *specific* programming language or paradigm, so an interesting question for us is if and how they could be exploited also in the context of agent-oriented programming. In this paper we focus in particular on error detecting, even if the notion of type introduced in simpAL (that will be presented in Section 3) has been devised to be of help also for improving organisation and optimisations of programs.

2.2 Detecting Errors in Current APLs

The support for (static) error detecting in current state-of-the-art APLs is quite limited, much weaker indeed compared to what we have e.g. in (statically) typed object-oriented programming languages.

Besides mere syntactical controls, there are APLs – e.g. Jason [2] – that do not provide any particular kind of checks, while others – such as 2APL [7], GOAL [11] and AFAPL [28] – provide some basic mechanisms for static errors detection. For example in 2APL warnings are generated when undefined *belief update actions* are referenced in the agent code. Similar controls are present in GOAL where a check is done about non-existing *user-defined actions* referenced in agent programs. For what concerns AFAPL instead, an old version of the language provides a quite rich set of static controls [3] (e.g. for incorrectly specified *activity identifiers*, for mistyped *imports*, etc.), nevertheless such controls have been removed – or just not re-implemented yet – in the current version of the language.

Overall, MAS developers are forced to deal *at runtime* with a set of programming errors that should be detected instead statically, before running the MAS program. In the following we provide some main examples of such programming errors, using a set of simple Jason source code snippets. We intentionally choose to consider samples written in only one APL just for making the description simple and terse. However, beside mere syntactical differences related to specific language constructs, through these samples we are able to outline a set of general considerations related to programming errors that do not hold only for Jason, but also apply to others state-of-the-art APLs.

```
1  // agent ag0
2  iterations("zero").
3
4  !do_job.
5
6  +!do_job
7    <-  ...
8        -+iterations(N+1);
9        ...
10       ?num_iterations(N).
11
12 +msgbel
13   <-  .print("Message received").
14
15 // agent ag1
16 !send_msg.
17
18 +!send_msg
19   <- .send(ag0, tell, msg_bel).
```

```
1  // agent ag2
2  !do_job.
3
4  +!do_job
5    <-  .send(ag3,achieve,floor_cleaned);
6        ...
7        !dojob.
8
9  //agent ag3
10 +!car_cleaned
11   <-  ...
```

```
1  // agent a4
2  iterations("zero").
3
4  +envPerceptA(ValueA)
5    <-  ...
6        actionA(10,20);
7        ?iterations(I)
8        actionA(10,I);
9        actionB(I);
10       actionB(10);
11       ?envPerceptC(ValueC);
12       nonExistingAction("hello").
13
14 +envPerceptC(Value)
15   <- ...
```

Fig. 1. Source code snippets showing a set of typical programming errors in Jason concerning: belief-related errors (on the left), goal-related errors (on top right) and agent-environment interaction errors (on bottom right)

First we consider issues related to beliefs, using the the snippet shown in Fig. 1 on the left. One of the most common belief-related errors concerns referencing non-existing beliefs in agent code, causing: *(i)* plan failures – e.g. line 10 where, due to a typo, we try to retrieve the belief `iterations(N)` using the predicate `num_iterations(N)` – and, *(ii)* the disabling of meaningful plans due to triggering events referring to non-existing perceivable events—e.g. the triggering event of the plan reported at lines 12-13 does not match the event generated by the reception of the message (`+msg_bel`) sent by agent `ag1` (line 19). Another beliefs-related issue concerns the possibility to write agent programs in which the same beliefs are bound to different value types in the course of agent execution. We argue that this can be problematic both from a conceptual viewpoint – i.e. a belief meant to be used for storing numeric information should not be used later also for storing strings literals – and also because such a permission can cause different runtime errors. For example the belief update action reported at line 8, being the belief `iterations` initialized with a string value (line 2), is not semantically correct and it hence produces, when executed, a runtime error.

We consider now issues related to goals, and in particular to goals assignment. It is possible to write correct MAS programs from a syntactical point of view, in which however wrong goals are assigned to agents at runtime, where wrong means e.g. goals that are unknown by the agents. Let's consider the case of agent `ag2` requesting to agent `ag3` the achievement of the goal

`floor_cleaned` (line 5 in Fig. 1, top right). Agent `ag3` is not able to achieve such a goal and the programmer can detect this issue only at runtime, by properly investigating why the MAS is not behaving as it is supposed to. As another example, the wrong goal self-assignment made by `ag2` (line 7 in Fig. 1, top right) – i.e. goal `!do_job` is referred as `!dojob` – is detected only at runtime when the agent realizes that it has no plan for dealing with the goal `!dojob`.

Finally we consider issues related to agent-environment interactions in agent programs. To this end we refer to the source code snippet reported in Fig. 1 on bottom right in which an agent `ag4` works in a classical Jason environment providing to the agent the external actions `actionA(<int>,<int>)` and `actionB(<String>)`; and generating percepts `envPerceptA(<int>)` and `envPerceptB(<String>)`. Even for what concerns agent-environment interactions it is quite simple to write source code that is correct from a mere syntactical perspective that however contains several errors from the semantic one. The source code reported in Fig. 1 on bottom right shows a set of the most common errors that can be made, and that can not be detected statically, when interacting with the environment in an agent program. In detail such errors are: *(i)* the invocation of environment actions providing arguments of the wrong type (e.g. line 8 and line 10), *(ii)* the invocation of non-existing environment actions (line 12), and *(iii)* the referencing of non existing percepts in both plan bodies (line 11) and in plan triggering events (line 14).

Some of the errors presented here – e.g. referencing a belief/goal that does not exists – may be detected statically quite easily, by enforcing the declaration of all the symbols in the MAS program in order to be effectively used. These errors are mainly related to the presence of typos, and they could be easily detected at compile time by constructing proper symbol tables to be used for the managing of symbols resolutions. For other kinds of errors instead – such as invoking an environment action with wrong arguments types, sending to an agent a message that exists but that the agent can not understand, etc. – the previous assumption is no longer sufficient. The introduction of typing would allow to detect even this kind of errors in a static manner, before running the MAS program.

3 Typing in simpAL

Before concentrating on the typing issue, first we give a brief overview of the main elements of the simpAL language. A prototype version of the simpAL platform – implemented in Java, including a compiler, an interpreter/virtual machine and an Eclipse-based IDE providing an editor with typical features such as context-assist, code completion, etc.[1] – is available for download as an open-source project[2], and can be used to test the examples discussed in this section. Because of lack of space, only those aspects of the language that are important

[1] Some snapshots of the IDE at work are available on the simpAL web site at `http://tinyurl.com/832o8hk`

[2] `http://simpal.sourceforge.net`

for this paper will be considered—the interested reader can refer to [26] and to the technical documentation on the web site for a more extensive account.

3.1 simpAL Overview

The main inspiration for simpAL abstractions comes – on the one side – from the A&A model [18] and from the BDI (Belief-Desire-Intention) model, in particular from its implementation in existing APLs, Jason in particular. On the other side, differently from existing BDI-based APLs, simpAL has been conceived conceptually as an extension of OOP languages with a further separated abstraction layer based on agent-oriented abstractions. The OOP layer – based on Java, but it could be any OOP language – is meant to be used solely to represent and manipulate abstract data types and data structures in general. All the other issues that, for instance, are related to concurrent programming (e.g. threads, synchronized methods, etc.) or I/O programming (e.g. network, GUI, OS related functionalities, etc.) are meant to be tackled using the agent-oriented abstraction layer.

By adopting a typical anthropomorphic and social view of computation, a simpAL program is given by an organization composed by a dynamic set of agents concurrently working in a shared, possibly distributed, environment. Agents are those components of the program (system) designed to perform autonomously *tasks*, that can be assigned both statically and dynamically to them. Autonomously means in this case that given a task to do, they pro-actively decide what actions to do and when to do them, promptly reacting to relevant events from their environment, fully encapsulating the control of their behavior. To perform their tasks, agents can create and use resources and tools, called generically *artifacts*. Artifacts are useful to represent those non-autonomous components of our program, the basic bricks composing the environment of the organization, providing some kind of functionality or service—such as easing agent communication and coordination (e.g. a blackboard), or interfacing agents with external environment or the user (e.g. a GUI, a socket), or wrapping external systems (e.g. a data-base, a web-service) or even simply helping agent work (e.g. a shared counter). An artifact can be used by a single agent or can be designed to be concurrently and safely used by multiple agents (e.g. a shared knowledge base, a shared calendar for alarms, etc.).

Agent interactions can occur in two basic ways that can be combined together: either indirectly through the environment (by using the same artifacts), or directly by means of asynchronous messages. In particular, agents have a basic set of communicative actions, that allow for sending messages either to inform or ask about some data or to assign/work with tasks. Agent-artifact interaction is based instead on the concept of *use* and *observation*, reminding the way in which artifacts are used by people in human environments. In order to be used, an artifact provides a set of *operations*, corresponding to the set of actions available to agents to use it. This implies that the repertoire of an agent's actions at runtime depends on the artifacts that the agent knows and can use. Besides operations, the usage interface of an artifact includes also *observable properties*,

as observable information concerning the dynamic state of the artifact which may be perceived and exploited by agents accordingly.

The overall (dynamic) set of agents and artifacts can be organized in one or multiple logical containers called *workspaces*, possibly in execution on different nodes of the network. An agent can also use – concurrently and transparently – artifacts located in different workspaces, not necessarily only those that belong to the workspace where the agent is running.

The computational model/architecture adopted for simpAL agents is a simplified version of the BDI one, implementing a sense-plan-act like execution cycle [26,27], but using OOP instead of logic programming to represent and manipulate data structures. An agent has a belief base, as a long term private memory storing information about: *(i)* the private state of an agent, *(ii)* the observable state of the environment, and *(iii)* information communicated by other agents. In simpAL the belief base is composed by a set of beliefs represented by simple variable-like information items, characterized by a name, a type, and a value—which could be any data object[3]. To perform tasks, an agent exploits the *plans* available in its plan library. Plans are modules of procedural knowledge specifying how to act and react to the events of the environment in order to accomplish some specific task. The set of plans in the plan library depends on the *scripts* loaded by the agent. As detailed later on, scripts are modules containing the description of set of plans, written by the agent programmers. An agent can handle multiple tasks in execution at a time.

3.2 Typing Agents with Tasks and Roles

In a software engineering perspective, a type defines a contract about what one can expect by some computational entity. In the case of objects, this concerns their interface, i.e. what methods can be invoked (and with which parameters) or – in a more abstract view – what messages can be handled by the objects. Conceptually, messages are the core concept of objects: receiving a message is the reason why an object moves and computes something. This is actually true also for active objects and actors.

Agents introduce a further level of abstraction. An agent does something because – first of all – it has a task to do (or rather a goal to achieve or maintain). It is quite intuitive then to define the type of an agent as its *contract* w.r.t. the organizational environment where it is immersed. In other words, conceiving the type of an agent as what one can expect by the agent in terms of the set of possible tasks that can be assigned to that agent. Following this idea we introduce the notion of *role* to explicitly define the type of an agent as the set of

[3] It is worth remarking that in existing agent-oriented languages beliefs are typically represented by first-order logic literals, denoting information that can be used by reasoning engines. However the logic representation is not necessarily part of the belief concept, as remarked by Rao and Georgeff in [23]: *"[beliefs] can be viewed as the informative component of the system state"* and *"[beliefs] may be implemented as a variable, a database, a set of logical expressions, or some other data structure"*([23], p. 313).

```
role Thermostat {

  task AchieveTemperature {
    input-params {
        targetTemp: double;
        threshold: double;
  }}

  task KeepTemperature {
    input-params {
      inputView: UserView;
    }
    understands {
      newThreshold: double;
  }}

  task DoSelfTest {
    talks-about {
      malfunctionDescr:
            MalfunctionInfo;
 }}}
```

```
usage-interface Conditioner {
  obs-prop isHeating: boolean;
  obs-prop isCooling: boolean;

  operation startHeating(speed: double);
  operation startCooling(speed: double);
  operation stop();
}
```

```
usage-interface Thermometer {
  obs-prop currentTemp: double;
}
```

```
usage-interface UserView {
  obs-prop desiredTemp: double;
  obs-prop threshold: double;
  obs-prop thermStatus:
        acme.ThermostatStatus;
}
```

Fig. 2. Definition of an agent role (on the left) and artifact interfaces (on the right)

the possible types of tasks that any agent playing that role is able to do. Fig. 2 on the left shows the definition of a role in simpAL. A role is identified by a name (e.g. `Thermostat`) and it includes the definition of the set of *task types*. A task type is identified by a unique identifier (name) inside the role. It defines a *contract* between the task *assigner* and *assignee*, in terms of a set of typed input/output parameters – `input-params` block and `output-params` block (not shown on this simple example) – and set of messages that can be understood by the task assignee – `understands` block – and the task assigner—`talks-about` block. A task type instance is like a record with the parameters assigned to some value. Typed attributes may contain any value/object of any Java class, plus also the identifiers of entities that are first-class simpAL abstractions, such as artifacts, agents, tasks, etc., which are typed too.

In simpAL information exchanges are always contextualized to tasks: so an agent A can send an information to another agent B only referring to a task instance t, without explicitly referring to B. A predefined action for exchanging messages among agents (`tell`) is provided:

```
/* the assigner tells a newThreshold msg to the assignee */
tell achieveTempTaskInstance.newThreshold = 100
/* the assignee tells a malfunctionDescr msg to the assigner */
tell doSelfTestTaskInstance.malfunctionDescr = new MalfunctionInfo(..)
```

The concept of role defining the agent type allows us to do error checking on: *(a)* the behavior of the agent implementing the role, checking that the agent implementation (the *how*) conforms to role definition (the *what*); *(b)* the behavior of the agents that aim at interacting with agents implementing a particular role, checking that: *(i)* they would request the accomplishment only of those tasks

that are specified by the role, and *(ii)* they would send only those messages that the tasks' assignee can understand.

Case *(a)* concerns performing two different controls when compiling *agent scripts*, which are the basic construct used to define agent concrete behavior (a brief description of agent scripts is reported in a separate box following Fig. 3). The first control is responsible of validating the script's plans w.r.t. the task types defined in the roles implemented by the script. The error checking rule states informally:

– for an agent script S, for each type of task T defined in any role R implemented by S, it must exist (at least) one plan P for T.

Given this rule, the `ACMEThermostat` script implementing the `Thermostat` role reported in Fig. 3 is correct, while a script like the following one:

```
1  agent-script IncompleteThermostatImpl implements Thermostat {
2    plan-for AchieveTemperature { ... }
3    plan-for DoSelfTest { ... }
4  }
```

would report an error message about missing a plan for a declared task, i.e. `KeepTemperature`.

The second control concerns checking messages that an agent playing certain roles tells to its tasks assigners (how assign a task to an agent is described below). This can be done by using the set of messages listed in the `talks-about` block of tasks definition. The checking rule in this case states:

– in a plan P related to a task type T, the messages sent by the *assignee* to the task *assigner* can only be the ones listed in T's `talks-about` block. In addition, the type of the messages sent must be compatible w.r.t. the message types defined in T.

Referring to the `ACMEThermostat` script, the only message that can be sent to tasks' assigners is the message `malfunctionDescr` in the context of the task type `DoSelfTest` (Fig. 3 line 66, where the prefix `this-task.` is used to identify the assignee's task instance—i.e, the task instance for which the plan will be instantiated at runtime), a task that can be used to check the correct functioning of the thermostat.

Case *(b)* concerns instead checking: *(i)* the assignment of tasks to agents playing a certain role R, and *(ii)* messages sent by a task assigner to the task assignee. Task assignment can be done in two ways.

```
1  assign-task taskInstanceTodo to: AgentId
2  do-task taskInstanceTodo task-recipient: AgentId
```

The first is through a predefined action `assign-task`. The action succeeds as soon as the task is successfully assigned to the assignee. It can also be used without specifying the target agent, so as for an agent to allocate the task to itself. The second is through a predefined action named `do-task`, which instead waits for the completion of the specified task instance—i.e. the action succeeds only when the task instance is successfully completed by the assignee.

```
agent-script ACMEThermostat implements Thermostat in SmartHome {

  savedThreshold: double

  plan-for AchieveTemperature  {
    #using: console@mainRoom, thermometer@bedRoom, conditioner@bedRoom

    println(msg: "Achieving temperature "
      + this-task.targetTemp + " from " + currentTemp);
    savedThreshold = this-task.threshold;
    {
      #completed-when:
        java.lang.Math.abs(this-task.targetTemp - currentTemp) < savedThreshold

      every-time currentTemp > (this-task.targetTemp + savedThreshold)
        && !(isCooling in conditioner) =>  startCooling(speed: 1) on conditioner
      every-time currentTemp < (this-task.targetTemp - savedThreshold)
        && !(isHeating in conditioner) =>  startHeating(speed: 1) on conditioner
    };
    stop()
  }

  plan-for KeepTemperature {
    #using: console@mainRoom, thermometer, conditioner, userView@mainRoom

    quitPlan : boolean = false;
    {
      #completed-when: quitPlan

      achiveTempTask: AchieveTemperature =
        new-task AchieveTemperature(targetTemp: desiredTemp in userView,
                                    threshold: threshold in userView);
      assign-task achiveTempTask

      every-time changed desiredTemp => {
        drop-task achiveTempTask;
        achiveTempTask = new-task AchieveTemperature(targetTemp: desiredTemp,
                                                     threshold: threshold);
        assign-task achiveTempTask
      }

      every-time changed currentTemp : !is-doing-any AchieveTemperature => {
        assign-task new-task AchieveTemperature(targetTemp: desiredTemp,
                                                threshold: savedThreshold)
      }

      every-time changed thermStatus
       : thermStatus.equals(acme.ThermostatStatus.OFF) => {
        if (isCooling || isHeating){
          stop()
        };
        drop-task achiveTempTask;
        quitPlan = true
      }

      every-time told this-task.newThreshold => {
        #atomic
        savedThreshold = this-task.newThreshold
      }
    }
  }

  plan-for DoSelfTest {
    ...
    if (someCondition) {
      tell this-task.malfunctionDescr = new MalfunctionInfo( ... )
    }
    ...
  }
}
```

Fig. 3. Definition of a script in simpAL

Defining Agent Scripts in simpAL (Fig. 3)

The behavior of an agent can be programmed in simpAL through the definition of scripts, that are loaded and executed by agents at runtime. Here we give a very brief account directly by using the `ACMEThermostat` example Fig. 3. The definition of an agent script includes the script name, an explicit declaration of the roles played by the script and then the script body, which contains the declaration of a set of *beliefs* and the definition of a set of *plans*. Beliefs in simpAL are like simple variables, characterized by a name, a type and an initial value. The `ACMEThermostat` script has just one belief, to keep track of the current threshold temperature to consider while doing its job. Beliefs declared at the script level are a sort of long-term memory of the agent, useful to keep track of information that could be accessed and updated by any plan in execution, and whose lifetime is equal to the one of the agent (script). Plans contain the recipe to execute tasks. The `ACMEThermostat` script has three plans, to achieve a certain temperature value (lines 5-21), to maintain a temperature value (lines 23-61), and to do some self test (lines 63-69). To do the `AchieveTemperature` task, the plan starts cooling or heating – using the conditioner – as soon as the current temperature is too high (lines 15-16) or too low (lines 17-18) compared to the target one (and the threshold)—the current temperature is observed by the thermometer. The information about the target temperature (`this-task.targetTemp`) derives from the related parameter of the task, while the belief about the current temperature (`currentTemp`) is related to the observable property of the `thermometer` artifact used in the plan. As soon as the current temperature is in the good range, the plan completes—stopping the conditioner if it was working. To do the `KeepTemperature` task, the plan achieves the desired temperature by immediately self-assigning the sub-task `AchieveTemperature` (line 33), which is executed also as soon as the desired temperature changes (lines 35-40) or the current temperature changes and the agent is not already achieving the temperature (line 42-45). The belief about the desired temperature (`desiredTemp`) comes from the observable property of the `userView` artifact used in the plan. Also, as soon as a message about a new threshold is told by the task assigner, the internal value of the threshold is updated (lines 56-59). The plan quits if the agent perceives from the `userView` artifact that the user has switched off the thermostat (lines 47-54). In that case, before quitting the plan, the conditioner is stopped if it was working. Finally, to do the `SelfTest` task, the agent performs some diagnostic operations (not reported in the sources) and if some malfunction condition is verified, a report containing the malfunction description is sent to the task assigner (line 66).

Explanations about some key elements of the syntax and semantics of plans follow—a more comprehensive description can be found here [26,27] and on simpAL technical documentation. The definition of a plan includes the specification of the type of task for which the plan can be used and a plan body, which is an action rule block. The action rule block contains the declaration of a set of local beliefs – that are visible only inside the block, as a kind of short-term memory – and a set of *action rules* specifying *when* executing *which* action. In the simplest case, an action rule is just an action and a block could be a flat list of actions. In that case, actions are executed in sequence, i.e. every action in the list is executed only after perceiving the event that the previous one has completed. In the most general case, an action rule is of the kind: `every-time | when` *`Event : Condition => Action`* meaning that the specified action can be executed every time or once that (`when`) the specified event occurs and the specified condition – which is a boolean expression over the agent beliefs base – holds. If not specified, the default value of the condition is true. Events concern percepts related to either one of *(i)* the environment, *(ii)* messages sent by agents or *(iii)* actions execution. All events are actually uniformly modeled as changes to some belief belonging to agent belief base, given the fact that observable properties, messages sent, and action state variables are all represented as beliefs. Furthermore, the syntax for specifying events related to a change of an observable property is `changed` *`ObsProp`* (e.g. line 35), the one for specifying the update of a belief about an information told by another agent is `told` *`What`* (e.g. line 56). If no event is specified, the predefined meaning is that the rule can be triggered immediately, but only once. Given that, the execution of a flat list of actions can be obtained by a sequence of action rules with only the action specified, separated by a semicolon (`;`). Actions can be: *(i)* external actions to affect the environment, i.e. operations provided by some artifact, *(ii)* communicative actions to directly interact with some other agent (to tell some belief, to assign a task, etc.), or *(iii)* predefined internal actions (to update internal beliefs, to manage tasks in execution, etc). An action can be also an action rule block {...}, which allows then to nest action blocks. Finally, the definition of an action rule block includes the possibility to specify some predefined attributes, for instance: the `#using:` attribute to specify the list of artifacts identifiers used inside the block (an artifact can be used/observed only if explicitly declared), the `#completed-when:` attribute to specify the condition for which the action rule block execution can be considered completed, the `#atomic` attribute to specify that the action rule block must be executed as a single action, without being interrupted or interleaved with blocks of other plans in execution (when the agent is executing multiple tasks at a time).

In both cases, we can enforce, statically, that:

– given a belief *Id* of type *R*, storing the identifier of some agent playing the role *R*, then for any action `assign-task t to: Id` or `do-task t task-recipient: Id`, there must exist a task type *T* in *R* such that *t* is a value (instance) of *T*. In case of task self-assignment the belief *Id* storing the agent identifier is implicit (it refers to the current agent).

Then, given a script fragment with a belief `myThermostat: Thermostat`, we have the following list of the main errors that can be caught at compile time:

```
1  /* compilation ok */
2  assign-task AchieveTemperature(targetTemp:21, threshold:2) to: myThermostat
3
4  /*  error: no tasks matching CleanTheRoom in role Thermostat */
5  do-task CleanTheRoom() task-recipient: myThermostat
6
7  /*  error: no targetT param in AchieveTemperature */
8  /*  error: missing threshold param */
9  assign-task AchieveTemperature(targetT: 21) to: myThermostat
10
11 /*  error: wrong type for the param value targetTemp */
12 /*  error: missing threshold param */
13 do-task AchieveTemperature(targetTemp: "21") task-recipient: myThermostat
```

The definition of a task type includes also the type of messages that the assigner can send to the task assignee. Given that, we can then check in agent scripts that the beliefs specified in the assigner's `tell` actions – those in which the task instance identifier is not `this-task.` – are among those listed in the `understands` block of the assignee role *R*, and that the types of the beliefs are compatible. In the example, when doing the task `KeepTemperature`, an agent playing the `Thermostat` role can be told about the new threshold to adopt – which is represented by the message `newThreshold` – by the assigner. Examples of checks follow:

```
1  keepTempTask: KeepTemperature
2  /* compilation ok */
3  tell keepTempTask.newThreshold = 2
4
5  /* error: aMsg is not listed in KeepTemperature understands block */
6  tell keepTempTask.aMsg = "hello"
7
8  /* error: wrong type for the belief newThreshold
9   * told to an agent playing the role Thermostat */
10 tell keepTempTask.newThreshold = "2"
```

Finally, some other kinds of errors can be checked in scripts at compile time thanks to the explicit declaration of beliefs (and their types): finding errors in plans about beliefs that are not declared neither as beliefs at the script level, nor as local beliefs of plans, nor as parameters of the task; or about beliefs that are assigned with expressions of wrong type.

3.3 Typing the Environment

On the environment side, we introduce the notion of *usage interface* defining the type of the artifacts, separated from its implementation provided by *artifact*

```
1  artifact ACMEConditioner implements Conditioner {
2    nTimesUsed: int;
3
4    init(){
5      isCooling = false; isHeating = true; nTimesUsed = 0;
6    }
7
8    operation startCooling(speed: double){
9      nTimesUsed++;
10     isCooling = true;  isHeating = false;
11     ...
12   }
13
14   operation startHeating(speed: double){...}
15
16   operation stop(){
17     isCooling = false; isHeating = false;
18     ...
19   }}
```

Fig. 4. Definition of an artifact template in simpAL. Artifact templates are used like classes in OOP, i.e. as templates to create instances of artifacts, defining then their internal structure and behavior. This figure shows the implementation of the toy `ACMEConditioner` artifact, implementing the `Conditioner` interface. The definition of a template includes the name of the template, the explicit declaration of the interfaces implemented by the template and then a body containing the declaration of the instance typed state variables of the artifact (e.g. `nTimesUsed`, line 2) – which are hidden, not observable – and the definition of operations' behavior. An operation is defined by a name (e.g. `startCooling`) (line 8), a set of keyword-based parameters (e.g. `speed`) and a body. The body is very similar to the one found in imperative OO languages – Java in this case is taken as main reference – so it is a block with a sequence of statements, including local variable declarations, control-flow statements, object related statements (object creation, method invocation, etc) and some pre-defined statements related to artifact functioning, that allow, for instance, for suspending the execution of the operation until some specified condition is met, or to terminate with a failure the operation execution.

templates. A usage interface is identified by a name and includes the specification of *(i)* the observable properties, and of *(ii)* the operations provided by all the artifacts implementing that interface—which correspond to the actions that agent can do on those kind of artifacts.

Fig. 2 on the right shows the definition of the artifacts used in the `ACMEThermostat` script, namely `Conditioner` – representing the interface of conditioner devices modeled as artifacts, used by agents to heat or cool – `Thermometer` – used by agents to be aware of the current temperature – and `UserView` – representing the interface of those GUI artifacts used to interact with the human users, in particular to know what is the desired temperature. Fig. 4 shows the skeleton of the definition of an artifact template implementing the `Conditioner` interface.

The introduction of an explicit notion of type for artifacts allows us to define a way to address two main issues: *(a)* on the agent side, checking errors about the actions (i.e. artifacts operations) and percepts (related to artifacts observable

state); *(b)* on the environment side, checking errors in artifact templates (i.e. the implementation), controlling that they conform to the implemented usage interfaces (i.e the type specification).

The case *(a)* concerns checking the action (rules) in plan bodies, so that: for each action *OpName*`(Params) on` *Target*, specified in an action rule, meaning the execution of an operation *OpName* over an artifact identifier *Target* whose type is *I*:

- there must exist an operation defined in the interface *I* matching the operation request;
- the action rule must appear in an action rule block (or in any of its parent block) where *Target* has been explicitly listed among the artifact used by the agent through the `#using:` attribute.

Examples of checks, given a fragment of a script with e.g. a belief `cond: Conditioner`:

```
1  /* compilation ok */
2  startCooling (speed: 1) on cond
3
4  /*  error: unknown operation switchOn */
5  switchOn () on cond
6
7  /*  error: unknown parameter time in startCooling operation */
8  startCooling (speed: 2 time: 10) on cond
9
10 /*  error: wrong type for the param value speed */
11 startCooling (speed: "fast") on cond
```

The target of an operation (e.g., `on cond`) can be omitted (as it happens in some points in plans of `ACMEThermostat` shown in Fig. 3) when there is no ambiguity with respect to the target of the artifacts that are currently used by the agent (specified in the `#using:` attribute).

On the event/percept side, we can check beliefs representing artifact observable properties in the event template of rules and in any expression appearing either in the context or in action rule body, containing such beliefs. For what concerns event templates, given an action rule: `updated` *Prop* `in` *Target* `: Context => Action`, where the event concerns the update of the belief about an observable property *Prop* in the artifact of type *I* denoted by *Target*, then the following checks apply:

- there must exist an observable property defined in *I* which matches *Prop*;
- the action rule must appear in an action rule block (or in any of its parent block) where *Target* has been explicitly listed among the artifacts used by the agent through the `#using:` attribute.

As in the case of operations, `in` *Target* can be omitted if there is no ambiguity about the artifact which is referred.

Examples of checks follow, supposing to have a fragment of a script with beliefs `cond: Conditioner` and `therm: Thermometer` about a conditioner and thermometer artifact:

```
1  /* compilation ok */
2  updated currentTemp => println(msg: "the temperature has changed")
3  updated currentTemp : isHeating
4          => println(msg: "the temperature has changed while heating...")
5  sum: double = currentTemp in therm + 1
6
7  /* error: unknown obs property isHeating  in Thermometer type */
8  updated isHeating in therm => ...
9
10 /* error: wrong type */
11 bak: boolean = currentTemp in therm
```

On the environment side (case *(b)*) the definition of the interface as a type allows for checking the conformance of artifact templates that declare to implement that interface, so that:

- for each operation signature *Op* declared in any of the interfaces *I* implemented by the template, the template must contain the implementation of the operation;
- for any observable property *Prop* that appears in expressions or assignments in operation implementation, then the declaration of the observable property must appear in one of the interfaces implemented by the template and the corresponding type expression must be compatible.

Finally, the explicit declaration of observable properties (in interfaces) and (hidden) state variables in artifact templates – the latter can be declared also as local variable in operations – allow for checking errors in the implementation of operations about the use of unknown observable properties/variables or about the assignment of values with a wrong type.

3.4 Typing the Overall Program Structure

In simpAL we use the notion of organization (recalling the human organization metaphor) to define the *main* of the overall multi-agent program. We introduce then the type of an organization, called *organization model*. An organization model is identified by a name and it used to explicitly define the workspace-based logic structure of the application. Besides the definition of its name, a workspace declaration in an organization model can include the explicit declaration of the identifiers (*literals*) of instances of artifacts and agents – along with their types – that are known to be available in that workspace[4]. Such identifiers are like global references that can be then referred in any agent script – so as to identify *"well-known"* agents to communicate with or artifacts to use – which explicitly declares to play a role *R* inside that organization model.

As a simple example, Fig. 5 (on the left) shows the definition of the `SmartHome` organization model, with: *(i)* a `mainRoom` workspace hosting the `userView` artifact and an agent `majordomo` of type `HomeAdmin`, and *(ii)* a

[4] In general, a workspace can contain at runtime also agents/artifacts not declared in the organization model: both can be dynamically created by agents by means of specific actions.

```
org-model SmartHome{

  workspace mainRoom {
    userView: UserView
    majordomo: HomeAdmin
  }

  workspace bedRoom {
    thermostat: Thermostat
    conditioner: Conditioner
    thermometer: Thermometer
  }}
```

```
org ACMESmartHome implements SmartHome{

  workspace mainRoom {
    majordomo = Majordomo()
                init-task: AdminHouse()
    userView = ACMEControlPanel()
  }

  workspace bedRoom {
    thermostat = ACMEThermostat()
    conditioner = ACMEConditioner()
    thermometer = ACMEThermometer()
  }}
```

Fig. 5. Example of the definition of an organization model in simpAL (on the left) and the a main organization file implementing the model (on the right)

`bedRoom` workspace hosting the remaining agents and artifacts. Given this organization model definition, then it is possible e.g. in the plan for the task `KeepTemperature` of the `ACMEThermostat` script (Fig. 3) to refer directly to the artifact `userView@mainRoom`.

Then a notion of *concrete organization* is introduced to define a concrete application instance, referring to an existing *organization model*. An example of organization definition is shown in Fig. 5 (on the right), sketching the definition of an `ACMESmartHome` concrete organization. A simpAL program in execution is a running instance of an organization.

The notion of organization model, defining the type for a simpAL organization, allows us to: *(a)* perform additional error checking controls in scripts *explicitly declared* in the context of an organization of a certain type, and *(b)* control that a concrete organization (i.e. the implementation) is conform w.r.t. its type specification (i.e. organization model). The case *(a)* allows to check, in those scripts sources declared inside an organizational context, that all the used literals refer to existing symbols defined in the related organization model.

On the organization side (case *(b)*), the definition of an organization model $OrgModel$ as a type allows for checking the conformance of a concrete organization instance Org that declares to implement that model, so that:

- each workspace Wsp declared in $OrgModel$ must be defined also in Org;
- each artifact literal $ArtLit$ of type I defined inside a workspace Wsp in $OrgModel$ must be must be correctly instantiated in the concrete organization Org. In particular such literal must be instantiated in Wsp, specifying an artifact template $ArtTempl$ implementing the usage interface I and, if needed, providing the initial parameters required by $ArtTempl$;
- each agent literal $AgLit$ of type R defined inside a workspace Wsp in $OrgModel$ must be must be correctly instantiated in the concrete organization Org. In particular such literal must be instantiated in Wsp, specifying an agent script $AgScript$ implementing the role R.

It is worth remarking that in the definition of a concrete organization Org implementing an organization model $OrgModel$, additional workspaces, agent and

artifact instances can be added to the ones initially declared in *OrgModel*. Examples of static checks that can be done follow, supposing to have a fragment of an organization that declares to implement the SmartHome organization model defined in Fig. 5.

```
org DummyHome implements SmartHome {
  /* compilation ok: new workspace */
  workspace newWsp {
    otherConsole = Console()
  }
  workspace bedRoom {
     /* error: missing instantiation of thermostat agent */
    conditioner = ACMEConditioner() /* compilation ok */
    /* error: wrong type. ACMEConditioner does
       not implement the required Thermometer role */
    thermometer = ACMEConditioner()
  }}
  /* error: missing mainRoom workspace */
```

As a final remark, the notion of organization used here is not meant to be as rich as the one that appears in MAS organization modelling. The main objective of introducing this concept here is to have a way to define rigorously the structure of the overall multi-agent program and to introduce some typing also at this level, in order to check errors at compile time related to the implementation of the overall program structure.

4 Related Work

As far as authors' knowledge, types and type systems have not received particular attention so far in the context of agent programming languages and agent-oriented programming.

In [10,9] a discussion about integrating algebraic data types, roles, and session types in the context of agent-oriented programming is sketched, starting from high-level similarities between certain aspects of an agent programming language (2APL) and a functional programming language (Haskell). Algebraic data types are used to constrain the content of messages; roles to constrain how particular agents interact, and sessions, to describe slices of the global interactions in the agent system. Together, these language features are introduced to support organisational concepts, as devised in agent-oriented methodologies and frameworks. Howver, the paper does not introduce an explicit notion of type for agents so as to improve static error checking.

Those agent-oriented platforms that are based or integrated with object-oriented languages / environments (e.g. JACK [14], Jade [1], Jadex [21]) can benefit of typing and static type checking provided by the lower-level OO layer (e.g. Java); however, such benefits are typically limited to the OO computational entities used in agent programs. So, it is not possible to detect at compile time errors related to e.g. the assignment of wrong tasks to agents or sending wrong messages—as far as authors' knowledge based on papers and official documentation.

Active Components are a recent development of the Jadex project, aiming at providing programming and execution facilities for distributed and concurrent systems [20]. The general idea is to consider systems to be composed of active (autonomous) components acting as service providers and consumers, following the Service Component Architecture (SCA) defined in the context of service oriented architectures. Communication among active components is preferably done then using service invocations, as defined by the service/component interfaces. Even if the approach does not clearly define a notion of type for agents (it is not its objective), this makes it possible to improve the kind of errors that could be detected at compile time by exploiting the service/component interfaces.

Besides sequential programming languages, type systems have been widely used for analyzing the behavior also of concurrent programs and systems of concurrent processes, to reason about deadlock-freedom, safe usage of locks, etc. [30,15]. In particular, the notion of *session type* has been introduced to specify complex interaction protocols, verified by static type checking [12]. Session types, and in particular multiparty session types [13], could be used to impose (and verify statically) restrictions on the pattern of interaction. These aspects are important indeed also in the context of programming languages based on agent-oriented abstractions and will be considered in our future work.

5 Concluding Remarks

The definition of a notion of type for agents, artifacts and organizations makes it possible to clearly separate the specification from the implementation, getting a first kind of substitutability. In particular, in every context of the program in which an agent playing some role R is needed, we can (re-)use any concrete agent equipped with a script – whose source code can be unknown, having only the compiled version – implementing the role R. Also, in every context of the program where an artifact providing the functionalities described by the I interface is needed, we can (re-)use any concrete artifact instance of an artifact template implementing the interface I. This enables a first level of reuse and evolvability, without the need of having the source codes. Improved version of agents and artifacts implementing some roles/interfaces can be introduced without doing any change in the other components that interact with them —if the roles and interfaces are not changed.

Indeed this is just a first step towards fully supporting the principle of substitutability, as defined in the context of OOP [29]. This requires the definition of a proper *subtyping* relationship, to define roles/interfaces as extensions of existing ones. This is part of our future work, exploring subtyping as a mechanism providing a sound and safe way to conceive the incremental modification and extension of agents/artifacts and their conceptual specialization.

Other important works in our agenda include: the definition of a proper formal model of the type system described in this paper – following a previous work introducing a core calculus for agents and artifacts [5] – so as to rigorously analyze its properties; and the improvement of typing for messages and communication

protocols, eventually exploiting results available both in agent-oriented programming literature and outside, such as the work on session types [8].

Finally, many of the concepts and abstractions on which simpAL is based can be found also in agent-oriented software engineering methodologies (an easy example is the very notion of role): these will be used then as a main reference for eventually refining and enriching how such concepts are currently modeled in simpAL.

References

1. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: Jade: A software framework for developing multi-agent applications. Lessons learned. Information & Software Technology 50(1-2), 10–21 (2008)
2. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd. (2007)
3. Collier, R.: Debugging agents in agent factory. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2006. LNCS (LNAI), vol. 4411, pp. 229–248. Springer, Heidelberg (2007)
4. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, pp. 125–135. ACM, New York (1990)
5. Damiani, F., Giannini, P., Ricci, A., Viroli, M.: A calculus of agents and artifacts. In: Cordeiro, J., Ranchordas, A., Shishkov, B. (eds.) ICSOFT 2009. CCIS, vol. 50, pp. 124–136. Springer, Heidelberg (2011)
6. Danforth, S., Tomlinson, C.: Type theories and object-oriented programmimg. ACM Comput. Surv. 20(1), 29–72 (1988)
7. Dastani, M.: 2apl: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
8. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
9. Grigore, C., Collier, R.: Supporting agent systems in the programming language. In: Hübner, J.F., Petit, J.-M., Suzuki, E. (eds.) Web Intelligence/IAT Workshops, pp. 9–12. IEEE Computer Society (2011)
10. Grigore, C.V., Collier, R.W.: Af-raf: an agent-oriented programming language with algebraic data types. In: Proceedings of the Compilation of the Co-located Workshops on DSM 2011, TMC 2011, AGERE! 2011, AOOPES 2011, NEAT 2011, & VMIL 2011, SPLASH 2011 Workshops, pp. 195–200. ACM, New York (2011)
11. Hindriks, K.V.: Programming rational agents in goal. In: Multi-Agent Programming, pp. 119–157. Springer US (2009)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284 (2008)
14. Howden, N., Rönnquist, R., Hodgson, A., Lucas, A.: JACK intelligent agents^{TM} — summary of an agent infrastructure. In: Proc. of 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS (2001)

15. Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003)
16. Meyer, B.: Static typing. In: ACM SIGPLAN OOPS Messenger, vol. 6, pp. 20–29. ACM (1995)
17. Mitchell, J.: Concepts in Programming Languages. Cambridge University Press (2002)
18. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems 17(3) (December 2008)
19. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
20. Pokahr, A., Braubach, L., Jander, K.: Unifying agent and component concepts: Jadex active components. In: Dix, J., Witteveen, C. (eds.) MATES 2010. LNCS, vol. 6251, pp. 100–112. Springer, Heidelberg (2010)
21. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.) Multi-Agent Programming. Kluwer (2005)
22. Rao, A.: AgentSpeak (L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
23. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: First International Conference on Multi Agent Systems, ICMAS 1995 (1995)
24. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agents and Multi-Agent Systems 23, 158–192 (2011)
25. Ricci, A., Santi, A.: Agent-oriented computing: Agents as a paradigm for computer programming and software development. In: Proc. of the 3rd Int. Conf. on Future Computational Technologies and Applications, Future Computing 2011, Rome, Italy. IARIA (2011)
26. Ricci, A., Santi, A.: Designing a general-purpose programming language based on agent-oriented abstractions: the simpAL project. In: Proc. of AGERE! 2011, SPLASH 2011 Workshops, pp. 159–170. ACM, New York (2011)
27. Ricci, A., Santi, A.: From actors to agent-oriented programming abstractions in simpal. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2012, pp. 73–74. ACM, New York (2012)
28. Ross, R., Collier, R., O'Hare, G.M.P.: AF-APL – bridging principles and practice in agent oriented languages. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 66–88. Springer, Heidelberg (2005)
29. Wegner, P., Zdonik, S.B.: Inheritance as an incremental modification mechanism or what like is and isn't like. In: Gjessing, S., Chepoi, V. (eds.) ECOOP 1988. LNCS, vol. 322, pp. 55–77. Springer, Heidelberg (1988)
30. Yoshida, N., Hennessy, M.: Assigning types to processes. Inf. Comput. 174(2), 143–179 (2002)