Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems

ProMAS @ AAMAS 2012

Organised by

Mehdi Dastani, Utrecht University, The Netherlands
Brian Logan, University of Nottingham, UK
Jomi F. Hübner, Federal University of Santa Catarina, Brazil

Held with The Eleventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)
June 5th, Valencia, Spain
Preface

Over the last decade, the ProMAS workshop series has provided a venue for state-of-the-art research in programming languages and tools for the development of multi-agent systems. ProMAS aims to address both theoretical and practical issues related to developing and deploying multi-agent systems, and the workshops have provided a forum for the discussion of techniques, concepts, requirements and principles central to multi-agent programming technology, including the theory and application of agent programming languages, the specification, verification and analysis of agent systems, and the implementation of social structures in agent-based systems. Many of these concepts and techniques have subsequently found widespread application in agent programming platforms and systems.

For the tenth edition of ProMAS, we are pleased to be able to present a programme of high quality papers covering a wide range of topics in multi-agent system programming languages, including language design and efficient implementation, agent communication and robot programming. We hope ProMAS 2012 will continue the successful tradition of previous ProMAS workshops, in contributing to the design of programming languages and tools that are both principled and at the same time practical for ‘industrial-strength’ multi-agent systems development.

ProMAS 2012 Workshop Organizers

Mehdi Dastani
Brian Logan
Jomi F. Hübner

June 2012
Organisation

Organising Committee

Mehdi Dastani Utrecht University, The Netherlands
Brian Logan University of Nottingham, UK
Jomi F. Hübner Federal University of Santa Catarina, Brazil

Steering Committee

Rafael H. Bordini Federal University of Rio Grande do Sul, Brazil
Mehdi Dastani Utrecht University, Netherlands
Juergen Dix Clausthal University of Technology, Germany
Amal E.F. Seghrouchni University of Paris VI, France

Program Committee

Natasha Alechina University of Nottingham, UK
Lacramioara Astefanoaei Laboratoire d’Informatique de Grenoble, France
Matteo Baldoni University of Torino, Italy
Olivier Boissier Ecole des Mines de St Etienne, France
Juan Botia Blaya University of Murcia, Spain
Lars Braubach University of Hamburg, Germany
Rem Collier University College Dublin, Ireland
Louise Dennis University of Liverpool, UK
Vladimir Gorodetsky Russian Academy of Sciences, Russia
Francisco Grimaldo Universitat de Valencia
James Harland RMIT, Australia
Koen Hindriks Delft University of Technology, The Netherlands
Benjamin Hirsch Technical University of Berlin, Germany
Max Knobbout Utrecht University, The Netherlands
Joao Leite University Nova de Lisboa, Portugal
Viviana Mascardi Genova University, Italy
Philippe Mathieu Univ. Lille 1, France
John-Jules Meyer Utrecht University, The Netherlands
Jorg Muller Clausthal University of Technology, Germany
Peter Novk Czech Technical University, Czech Republic
Alexander Pokahr University of Hamburg, Germany
Alessandro Ricci  
University of Bologna, Italy

Birna van Riemsdijk  
Delft University of Technology, The Netherlands

Ralph Ronnquist  
Intendico, Australia

Sebastian Sardina  
RMIT University, Australia

Ichiro Satoh  
NII, Japan

Michael Ignaz Schumacher  
University of Applied Sciences Western Switzerland, Sierre, Switzerland

Kostas Stathis  
Royal Holloway, University of London, UK

Bas Steunebrink  
University of Lugano, Switzerland

John Thangarajah  
RMIT, Australia

Pankaj Telang  
Cisco

Wamberto Vasconcelos  
University of Aberdeen, UK

Jørgen Villadsen  
DTU Informatics, Denmark

Neil Yorke-Smith  
American University of Beirut, Lebanon and SRI International, USA

Yingqian Zhang  
Delft University of Technology, The Netherlands

Additional Reviewers

Antonn Komenda
Matthias Knorr
Andrea Santi
Table of Contents

eJason: an implementation of Jason in Erlang ................................. 7
Álvaro Fernández Díaz, Clara Benac Earle, and Lars-Ake Fredlund

Conceptual Integration of Agents with WSDL and RESTful Web
Services .............................................................. 23
Lars Braubach and Alexander Pokahr

Agent Programming Languages Requirements for Programming
Cognitive Robots .......................................................... 39
Pouyan Ziafati, Mehdi Dastani, John-Jules Meyer, and Leendert van
der Torre

An Agent-Based Cognitive Robot Architecture ............................... 55
Changyun Wei and Koen V. Hindriks

A Programming Framework for Multi-Agent Coordination of Robotic
Ecologies ................................................................. 69
M. Dragone, S. Abdel-Naby, D. Swords, and M. Broxvall

Evaluation of a Conversation Management Toolkit for Multi Agent
Programming ............................................................... 85
David Lillis, Rem W. Collier, and Howell R. Jordan

Compact and Efficient Agent Messaging ...................................... 101
Kai Jander and Winfried Lamersdorf

Query Caching in Agent Programming Languages ......................... 117
Natasha Alechina, Tristan Behrens, Koen V. Hindriks, and Brian
Logan

Typing Multi-Agent Programs in simpAL .................................. 132
Alessandro Ricci and Andrea Santi

Adaptive Programming in GOAL ........................................ 148
Dhirendra Singh and Koen V. Hindriks

Author Index ............................................................... 164
eJason: an implementation of Jason in Erlang

Álvaro Fernández Díaz, Clara Benac Earle, and Lars-Åke Fredlund
Babel Group. Universidad Politécnica de Madrid, Spain
{avalor,cbenac,fred}@babel.ls.fi.upm.es

Abstract. In this paper we describe eJason, a prototype implementation of Jason, the well-known agent-oriented programming language, in Erlang, a concurrent functional programming language. The reason for choosing Erlang as the implementation vehicle is the surprising number of similarities between Jason and Erlang, e.g., both have their syntactical roots in logic programming, and share an actor-based process and communication model. Moreover, the Erlang runtime system implements lightweight processes and fast message passing between processes. Thus, by mapping Jason agents and agent-to-agent communication onto Erlang processes and Erlang process-to-process communication, we can create a very high-performing Jason implementation, potentially capable of supporting up to a hundred thousand concurrent actors. In this paper we describe in detail the implementation of Jason in Erlang, and provide early feedback on the performance of the implementation.

1 Introduction

Among the different agent-oriented programming languages, AgentSpeak [13] is one of the most popular ones. It is based on the BDI architecture [14, 17], which is central in the development of multiagent systems. AgentSpeak allows the implementation of rational agents by the definition of their know-how, i.e. how each agent must act in order to achieve its goals. AgentSpeak has been extended into a programming language called Jason [7, 9]. Jason refers to both the AgentSpeak language extension and the related interpreter that allows its execution in Java. Thus, Jason is an implementation of AgentSpeak that allows the construction of multiagent systems that can be organized in agent infrastructures distributed in several hosts. It allows the interfacing to the JADE Framework [5, 6], thus generating multiagent systems fully compliant to FIPA [1, 12] specifications. To effortlessly distribute the agent infrastructure over a network, the use of the SACI [11, 2] middleware is suggested. Jason has been designed to address the desirable properties of rational agents identified in [16]: autonomy, proactiveness, reactivity and social ability. In the rest of the paper we assume that the reader is familiar with Jason [9].

A significant new trend in processor architecture has been evident for a few years. No longer is the clock speed of CPUs increasing at an impressive rate,
rather we have started to see a race to supply more processor elements in mainstream multi-core CPU architectures coming from Intel and AMD. Initially, the software industry has been slow in reacting to this fundamental hardware change, but today, utilising multiple cores is the only way to improve software system performance. With traditional programming languages (such as Java, C, C++, etc.) writing bug-free concurrent code is hard, and the complexity grows quickly with the number of parallel tasks. As a result, alternative languages, with less error-prone concurrency primitives, are attracting more attention.

Following this trend, the Erlang programming language [3, 10] is gaining momentum. The usage has increased, and among the users are large organisations like Facebook, Amazon, Yahoo!, T-Mobile, Motorola, and Ericsson. The most prominent reasons for the increased popularity of Erlang are lightweight concurrency based on the actor model, the powerful handling of fault tolerance, the transparent distribution mechanisms, the generic OTP design patterns, and the fact that the language has functional programming roots leading to a small, clean code base.

In this paper we report on our experience translating the Jason programming language to Erlang. The similarities between Jason and Erlang – both are inspired by Prolog, both support asynchronous communication among computational independent entities (agents/processes) – make the translation rather straightforward. By implementing Jason in Erlang we offer the possibility to Erlang programmers of using an agent-oriented programming language like Jason integrated in Erlang. To Jason programmers, the approach gives them the possibility of executing their code in the Erlang runtime system, which is particularly appropriate for running robust multiagent systems with a large number of concurrent and distributed agents.

Moreover, as the syntax of Erlang is inspired by Prolog ¹, e.g., having atoms beginning with a lowercase letter, and single-assignment variables beginning with an uppercase letter, etc., we hope to reduce the conceptual gap for a Jason programmer interested in modifying the Jason meta-level (e.g., changing the selector functions, and implementing actions) by adopting Erlang, compared to having to use Java. Perhaps even more interesting is the potential for introducing Erlang programmers to the world of BDI programming through this new Jason implementation. This is a group of programmers already used to thinking of programming systems composed of independent communicating agents (or in the terminology of Erlang, processes), and superficially familiar with the syntax of Jason. To us it appears that the conceptual gap between programming agents in Jason, and functions and processes in Erlang, is smaller than for many other programming languages (Java).

A prototype of the implementation of Jason in Erlang is available at

\[\textit{git} : //github.com/avalor/eJason.git\]

The rest of the paper is organized as follows. Before explaining the translation, the main characteristics of Erlang are briefly described in Sect. 2. Then,

¹ Not surprisingly, as the first implementation of Erlang was written in NU Prolog [4]
in Sect. 3 the translation of the Jason constructs, the Jason reasoning cycle, the process orchestration of eJason, and the current limitations of the approach are explained. Some early benchmarks results for the eJason prototype are reported in Sect. 4. Finally, a summary of our conclusions and items for future work appear in Sect. 5.

2 Erlang

Erlang [3, 10] is a functional concurrent programming language created by Ericsson in the 1980s. The chief strength of the language is that it provides excellent support for concurrency, distribution and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. It enables programmers to write robust and clean code for modern multiprocessor and distributed systems. In this section we briefly describe the key aspects of Erlang.

2.1 Functional Erlang

In Erlang basic values are: integers, floats, atoms (starting with a lowercase letter), bit strings, binaries, and funs (to create anonymous functions), and process identifiers (pids). The compound values are lists and tuples. Erlang syntax includes a record construct which provides syntactic sugar for accessing the elements of a tuple by name, instead of by position. Functions are first class citizens in Erlang. For example, consider the declaration of the function factorial that calculates the factorial of a number.

```erlang
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

As in Prolog, variable identifiers (N) start with a capital letter, and atoms (factorial) with a lowercase letter. Like Prolog, Erlang permits only single assignment to variables.

As virtually all functional programming languages, Erlang supports higher order functions.

2.2 Concurrent and Distributed Erlang

An Erlang system (see Fig. 1) is a collection of Erlang nodes. An Erlang node (or Erlang Run-time System) is a collection of processes, with a unique node name. Communication is asynchronous and point-to-point, with one process sending a message to a second process identified by its pid. Messages sent to a process are put in its message queue, also referred to as a mailbox. Informally, a mailbox is a sequence of values ordered by their arrival time. Mailboxes can in theory store any number of messages. Although mailboxes are ordered, language constructs permit retrieving messages from the process mailbox in arbitrary order.

As an alternative to addressing a process using its pid, there is a facility for associating a symbolic name with a pid. The name, which must be an atom,
is automatically unregistered when the associated process terminates. Message passing between processes in different nodes is transparent when pids are used, i.e., there is no syntactical difference between sending a message to a process in the same node, or to a remote node. However, the node must be specified when sending messages using registered names, as the pid registry is local to a node.

A unique feature of Erlang that greatly facilitates building fault-tolerant systems is that processes can be “linked together” in order to detect and recover from abnormal process termination. If a process $P_1$ is linked to another process $P_2$, and $P_2$ terminates with a fault, process $P_1$ is automatically informed of the failure of $P_2$. It is possible to create links to processes at remote nodes.

As an integral part of Erlang, the OTP library provides a number of very frequently used design patterns (behaviours in Erlang terminology) for implementing robust distributed and concurrent systems. The most important OTP design patterns are generic servers that implement client/server architectures, and supervisors, to build robust systems. Other OTP design patterns implement, for instance, a publish-subscribe mechanism, and a finite state machine.

3 Implementing Jason in Erlang

This section describes the implementation of a subset of Jason in Erlang.

3.1 A simple running example in Jason

To illustrate the implementation of Jason in Erlang, we use the example in Fig. 2, which illustrates the main syntactical elements of the Jason language.

This somewhat artificially programmed agent is a counter, which prints a message when finished. The initial beliefs of the agent are (a), representing that
the agent believes the initial value to be zero, and (b), representing that the agent believes that it has to count up to 2000. There is one rule (c), expressing the successor relation for numbers. The agent’s initial goal (d) is to start counting.

There are three plans (e), (f) and (g). Plan (e) initializes the actual counter by adding a new belief to the agent’s belief base and introduces a new achievement goal ![count](1). That goal can be achieved by plans (f) and (g), whose context’s are disjoint and, thus, can never be considered as applicable plans at the same time. When plan (g) is executed, which occurs when the agent has counted up to its limit, it prints a message and the agent remains waiting as there are no more events. We kindly direct the reader to [9] for a complete definition of the Jason programming language and its interpreter, as a detailed description of the different features of Jason lies beyond the scope of this paper.

### 3.2 An overview of the implementation

Jason is both a programming language which is an extension of AgentSpeak, and an interpreter of this programming language in Java. The constructs of the Jason programming language can be separated into three main categories: beliefs, goals and plans. The Jason interpreter runs an agent program by means of a reasoning cycle that provides the operational semantics of the agent. This semantics has been formalised and can be found in [9].

The translation of beliefs and goals to Erlang is rather straightforward since they represent the knowledge of an agent, rather than its behaviour (with the exception of rules). Common Erlang data types and functions are used to translate these Jason constructs to Erlang. Initially we used the third party software
ERESYE [15] (ERlang Expert SYstem Engine) to implement the belief base of each agent. ERESYE is a library to write expert systems and rule processing engines using the Erlang programming language. It allows to create multiple engines, each one with its own facts and rules to be processed. We decided to use this software as the term storage service due to its capabilities to store Erlang terms and to also retrieve them using pattern matching. Nevertheless, due to the way in which we used this software, the resulting Jason implementation was rather inefficient. Therefore we decided to implement our own belief base. This later implementation represents the belief base of each agent as a list of ground terms. The translation of beliefs, goals and rules to Erlang is explained in Sect. 3.3.

The implementation of plans is more convoluted due to their dynamic nature. Every plan is composed by one or more formulas that must be evaluated sequentially. However, the formulas in a plan may not all be executable in the same reasoning cycle. The representation of plans in Erlang, and their execution by a tail-recursive Erlang function, is explained in Sect. 3.4.

A higher-level view of the different Erlang processes implementing the Jason reasoning cycle [9] and the communication between them is described in Sect. 3.5, while Sect. 3.6 provides the details. Basically, the reasoning cycle of each agent is handled by a different Erlang process.

Finally Sect. 3.7 enumerates the limitations of eJason, with respect to implementing the full Jason language, at the time of writing this paper.

3.3 Translation of Jason beliefs and goals into Erlang

Here we describe how the different constructs for representing and inferring knowledge of Jason are implemented.

**Variables.** To represent the bound and unbound variables of a plan we use a variable valuation that is updated as variables become bound to values. Concretely, a valuation for a plan is represented by an Erlang tuple where values are associated with distinct variables ordered according to the order in which these variables first occur in the plan. For instance, a possible valuation for the second plan (ε) in Fig. 2 would be \{0, 2000, '_'\}, thus binding X to 0, Y to 2000 and leaving NewCount unbound.

**Beliefs.** Every agent possesses its own belief base, i.e., each agent can only access and update its own belief base. In a first version of eJason, we used ERESYE in the following manner. Each agent ran its own ERESYE engine, which spawned three Erlang processes for each belief base. Early experiments showed that this implementation was rather inefficient. For instance, the eJason implementation of the counter example could only handle around four thousand agents. An alternative is to use a single ERESYE engine for all agents, and provide some means to isolate the beliefs of each agent from everyone else’s. We discarded this approach because the autonomy of agents would have been compromised.
For instance, a failure in the ERESYE engine would cause a failure in the belief base of all agents. Therefore, we decided to implement our own belief base in a separate module, named beliefbase, which provides the functionality to access and update a belief base without having to create a separate Erlang process. As explained earlier, this belief base is represented as a list of Erlang terms, where each term in the list corresponds to a different belief.

A belief, i.e., either an atom or a ground formula, is represented in eJason as an Erlang tuple. An atom belief is represented in Erlang as the tuple containing the atom belief itself, e.g., {atom_belief}. A ground formula belief is represented by an Erlang tuple with three elements. The first element is the name of the predicate, the second is a tuple which enumerates the arguments of the predicate, and the third is a list containing a set of annotations. Each annotation can either be an atom or a predicate and is represented in the same manner as a belief. As an example, the belief base of the running example (with an added annotation):

```
init_count(0).
max_count(2000)[source(self)].
```

is translated to the following Erlang term:

```
{ init_count, {0}, [] }.
{ max_count, {2000}, [ {source,{self},[]} ] }.
```

**Rules.** Each rule in Jason is represented as an Erlang function. This function, when provided with the proper number of input parameters, accesses the belief, if necessary, and returns the list of all the terms that both satisfy the rule and match the input pattern.

**Goals.** Goals are represented in the same way as beliefs. Nevertheless, they are never stored in isolation, but as part of the body of an event, as specified below.

**Events.** As we have not yet implemented perception of the agent environment, events always correspond to the explicit addition or deletion of beliefs, or the inclusion of achievement and test goals. An event is composed of an event body, an event type, and a related intention. The event body is a tuple that contains two elements. The first element is one of the atoms {added_belief, removed_belief, added_achievement_goal, added_test_goal}. The second element is a tuple that represents the goal or belief whose addition or deletion generated the event. The event type is either the atom internal or the atom external, with the obvious meaning. The related intention is either a tuple, as described below, or the atom undefined to state that the event has no related intention. The only internal events that possess a related intention are the events corresponding to the addition of goals, as their intended means will be put on top of that intention. The intended means for the rest of events will often generate new intentions. When a relevant plan for the event is selected, the list of Erlang functions that
execute the formulas in its body is added either on top of a related intention or as a brand new intention (e.g. in the case of external events). For instance, consider the following formulas in the body of a plan belonging to some intention $Intention$:

$$+\text{actual_count}(NewCount);$$

$$?\text{next}(X, NewCount);$$

The events generated after their respective execution would be:

$$\{\text{event, internal, \{added\_belief,}$$

$$\{\text{actual\_count,\{NewCount}, []\}}\}, \text{undefined}\}$$

$$\{\text{event, internal, \{added\_test\_goal,}$$

$$\{\text{next, \{X, NewCount}, []\}}\}, \text{Intention}\}$$

For the sake of clarity, the variable $Intention$ appears as placeholder for the real representation of the corresponding related intention.

### 3.4 Implementing Jason plans in Erlang

**Body of a plan.** Every Jason plan is composed by one or more formulas that must be evaluated in a sequence. However, these formulas are not all necessarily evaluated during the same reasoning cycle of the agent, e.g., due to the presence of a subgoal that must be resolved by another plan. To be able to execute the formulas separately, each formula is implemented by a different Erlang function. Then, the representation of the body of a Jason plan is a list of Erlang functions. Each of these functions implements the behaviour of a different formula from the Jason plan. The order of these functions in the list is the same order of the body formulas they represent. The implementation and processing of the formulas in a plan body is the most intricate task in the implementation of Jason in Erlang.

**Plans.** A Jason plan is represented by a record having three components: a *trigger*, a *context* and *body*. The trigger element is a function which, applied to the body of an event, returns either the atom *false* if the plan does not belong to the set of relevant plans for the event or the tuple $\{true, InitialValuation\}$, where $InitialValuation$ provides the bindings for the variables in the trigger. The context is a function which, when applied to the initial valuation obtained from the trigger, returns a list of all the possible valuations for the variables in the trigger and context that satisfy the context. Finally, the body element is the list of Erlang functions that implement the body of the plan, as described before.

As an example, consider the plan for agent *counter*:

$$+!\text{startcount} : \text{init\_count}(X) \leftarrow +\text{actual\_count}(X);$$

$$!\text{count}.$$  

The *plan* record generated for the plan above is
{plan, fun start_count_trigger/1,
   fun start_count_context/1,
   [Fun1, Fun2]}

where the start_count_trigger/1 and start_count_context/1 functions implement the trigger and the context respectively. The list at [Fun1,Fun2] represents the plan body, where Fun1 implements the formula +actual_count(X) and Fun2 implements the formula !count.

**Intentions.** The stack of partially instantiated plans that compose each of the Jason intentions is represented as a list of Erlang records. Each of these records is composed of four elements. The first element is the event that triggered the plan. This element is kept as a meta-level information that can be accessed by the intention selection function. For instance, we could give priority to the execution of intentions whose partially instantiated plan on top of the stack resolves a test goal. The second element is the plan record chosen by the option selection function and, again, is intended to serve as a meta-level information accessible by the intention selection function. The third element is a tuple that represents the intended means of the intention plan, i.e. the bindings for the variables in the partially instantiated plan. The fourth element is a list of Erlang functions, representing the formulas of the partially instantiated plan that have not been executed yet. If an intention is selected for execution, the record for the partially instantiated plan on top of it (i.e. the first element of the list that represents the intention) is obtained. Then, the function at the head of the list of Erlang functions in the fourth element of this record is applied to the current variable valuation. Finally, this last function is removed from the list. This process amounts to processing the formula on top of the intention stack as is required by the specification in [9].

**Selection functions.** The event, plan and intention selection functions for a MAS can be customised by providing new implementations (in Erlang) of the functions selectEvent, selectPlan and selectIntention.

### 3.5 Process Orchestration and Communication

The multiagent system generated by the translation from Jason to Erlang maps each agent to an Erlang process, all executing on the same Erlang node. Each Erlang agent process can be accessed using either its process identifier, or the name of the Jason agent. The name of the agent is associated with the Erlang process using the Erlang process registry. In case multiple agents are created with the same name an integer (corresponding to the creation order) is appended to the registered name to keep such names unique. An item for future work is to extend Jason with new mechanisms to create multiple agents from the same agent definition, and to associate symbolic names with such agents, as the present mechanisms are somewhat unwieldy.
The communication between agents is implemented using Erlang message passing. As an example, consider a system where the agent *alice* sends different messages to agent *bob* by executing the internal action formulas:

```erlang
.send(bob, tell, counter(3))
.send(bob, untell, price(coffee,300))
.send(bob, achieve, move_to(green_cell))
```

The actions are mapped to the following Erlang expressions:

- `bob ! {communication,alice,{tell,{counter,{3},[]}}}.`
- `bob ! {communication,alice,{untell,{price,{coffee,300},[]}}}.`
- `bob ! {communication,alice,{achieve,{move_to,{green_cell},[]}}}.`

The Erlang expression `Receiver ! Message` deposits `Message` into the mailbox of agent `Receiver`. The atom `communication` is used to declare the message type. It is included in the implementation to enable processes to exchange other types of messages, possibly not related to agent communication, in a future extension of eJason.

Agent *bob* can process the different messages sent by *alice* by checking its mailbox, which is performed automatically in every iteration of the reasoning cycle. The Erlang expression that retrieves the message from the process mailbox:

```erlang
receive {communication,Sender,{Ilf,Message}} ->
case Ilf of
tell -> ... %% Process tell message
untell -> ... %% Process untell message
achieve -> ... %% Process achieve message
end.
```

These examples show how easily the agent communication between Jason agents can be implemented in Erlang. The simple yet efficient process communication mechanism of Erlang is one of the principal motivations to implement Jason agents using the Erlang programming language.

In the example above, all the agents are located in the same MAS architecture; messaging between agents in different architectures would be easy to support too, and would not require the use of a communication middleware like SACI. However, such an extension is not yet implemented in eJason.

### 3.6 Representing the Jason Reasoning Cycle in Erlang

The Jason reasoning cycle [8] must of course be represented in eJason. We implement the reasoning cycle using an Erlang function `reasoningCycle` with a
single parameter, an Erlang record named `agentRationale`, which represents the current state of the agent. The elements of this record are: an atom that specifies the name of the agent, a list that stores the events that have not yet been processed, the list of executable intentions for the agent, the list of executable plans, a list of the terms that compose the agent belief base, and three elements (`selectEvent`, `selectPlan` and `selectIntention`) bound to Erlang functions implementing event, plan and intention selection for that particular agent (in this manner each agent can tailor its selection functions; appropriate defaults are provided).

Below a sketch of the `reasoningCycle` function is depicted, providing further details on how eJason implements the reasoning cycle of Jason agents:

```erlang
reasoningCycle(OldAgent) ->
    Agent = check_mailbox(OldAgent),
    #agentRationale
    {events = Events,
     belief_base = BB,
     agentName = AgentName,
     plans = Plans,
     intentions = Intentions,
     selectEvent = SelectEvent,
     selectPlan = SelectPlan,
     selectIntention = SelectIntention} = Agent,

    {Event,NotChosenEvents} = SelectEvent(Events),

    IntendedMeans =
        case Event of
            [] -> [];  %% No events to process
            _ ->
                RelevantPlans = findRelevantPlans(Event,Plans),
                ApplicablePlans = unifyContext(BB, RelevantPlans),
                SelectPlan(ApplicablePlans)
        end,

    AllIntentions = % The new list of intentions is computed
                  processIntendedMeans(Event,Intentions,IntendedMeans),

    case SelectIntention(AllIntentions) of
        {Intention,NotChosenIntentions} ->
            Result = executeIntention(BB,Intention),
            NewAgent =
                applyChanges
                (Agent#agentRationale
                 {events = NotChosenEvents,
                  intentions = NotChosenIntentions},
                 Result),
            reasoningCycle(NewAgent);
        end.
```

This record is updated during the execution of each reasoning cycle:

1. At the beginning of each reasoning cycle, the agent checks its mailbox and processes its incoming messages, adding new events.
2. The event selection function included in the \textit{agentRationale} record is applied to the list of events also included in the same record. The result of the function evaluation is an Erlang record of type \textit{event}. This record represents the unique event that will be processed during the current reasoning cycle.
3. The function trigger of every plan is applied to the body of the selected event. For every distinct valuation returned by a trigger function, a new plan is added to the list of relevant plans. Each relevant plan is represented by a \textit{plan} record along with a valuation for the parameter variables.
4. Next, the context function of each relevant plan is evaluated. The result of each function application is either an extended valuation, possibly binding additional variables, or the failure to compute a valuation that is consistent with both the trigger and the context. For each remaining valuation, a new plan is added to the set of applicable plans. Each applicable plan is represented by a set of variable bindings along with a \textit{plan} record.
5. The plan selection function is applied to the list of applicable plans. The result obtained is an applicable plan that represents the new intended means to be added to the list of intentions.
6. The intention selection function is applied to the list of executable intentions. It selects the intention that will be executed in the current reasoning cycle. Note that, as specified by the Jason formal semantics, this intention may not necessarily be the intention that contains the intended means for the event processed at the beginning of the reasoning cycle.
7. The first remaining formula of the plan that is at the head of the chosen intention is evaluated. The result of evaluating a function may generate new internal or external events, e.g. by adding a new belief to the belief base.
8. The new events generated are added to the list of events stored in the \textit{agentRationale} record representing the state of affairs of the agent. If the formula evaluated was the last one appearing in a plan body, the process implementing the plan body terminates. If, moreover, the plan that finished was the last remaining plan in the corresponding intention, the intention itself is removed from the list of executable intentions.
9. Finally a new reasoning cycle is started by repeating steps 1-9 with the new updated \textit{agentRationale} record.

3.7 Jason Subset Currently Supported

eJason currently supports only a subset of the Jason constructs needed to implement complex multi-agent systems. However, we foresee no major difficulties in adding the additional features not currently supported, and expect to do so in the near future. The features of the Jason language not currently supported are the following:
1. **Belief annotations.** Even though our Jason parser accepts code with belief annotations, these annotations are not taken into account when resolving plans (e.g., when checking whether a plan context is satisfied).

2. **Annotations on plan labels.** The meta-level information associated with plans is removed during the lexical analysis.

3. **Plan failure handling.** Whenever a plan fails, e.g., because test goal in the plan body cannot be successfully resolved, the whole intention that the plan belongs to is dropped. Moreover, no new event is generated as a result of the plan failure.

4. **Environment.** The environment of eJason programs is not currently modelled. Therefore, no external actions, except console output, are allowed and no perception phase is required.

5. **Distribution.** There is no support for distributed agents.

6. **Communication.** The only illocutionary forces that are properly processed are *tell*, *untell* and *achieve*. Messages with any other kind of illocutionary force are ignored and dropped from the mailbox of the agent.

7. **Library of internal actions.** The only internal actions considered are “.print” (to display text on the standard output) and “.send” (to interact with other agents in the same multiagent system).

8. **Unbound plan triggers.** The trigger of every plan must be either an atom or a predicate (whose parameters do not need to be bound) but never a variable.

9. **Decomposition operator.** The binary operator “=..”, used to (de)construct literals (i.e. predicates and terms), is not accepted by the parser.

10. **Code order.** The grammar accepted by the parser is similar to the simplified one presented in Appendix 1 of [9]. Therefore, the source code to be translated must state first the initial beliefs and rules, followed by the initial goals and, finally, the different plans.

11. **Multiagent system architecture.** There is only one kind of agent infrastructure implemented. It runs all the agents in a multiagent system within the same Erlang node.

4 Experiments

To test the performance of eJason, we use two simple Jason programs. The first is the counter example of Fig. 2 in Section 3.1. The second represents an agent that outputs two greeting messages on the console. To add some complexity to the behaviour, the contents of those messages are obtained from the set of beliefs of the agent using queries which have both bound and unbound variables. The examples were run using different numbers of homogeneous agents, i.e., all the agents behaved the same. All of them were run under Ubuntu Linux version 10.04 in a computer with two 2.53 GHz processors. With these examples, we want to measure the execution time of the generated MAS and their scalability with respect to the number of agents in the system.

The preliminary results are presented in Tables 1 and 2.
<table>
<thead>
<tr>
<th>Number of Agents</th>
<th>Jason Execute Time (magnitude)</th>
<th>eJason Execution Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>milliseconds</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>milliseconds</td>
<td>46</td>
</tr>
<tr>
<td>1000</td>
<td>seconds</td>
<td>181</td>
</tr>
<tr>
<td>10000</td>
<td>minutes</td>
<td>1916</td>
</tr>
<tr>
<td>100000</td>
<td>not measurable</td>
<td>18674</td>
</tr>
<tr>
<td>5000000</td>
<td>not measurable</td>
<td>97086</td>
</tr>
<tr>
<td>80000000</td>
<td>not measurable</td>
<td>165522</td>
</tr>
</tbody>
</table>

**Table 1.** Execution times for the counter multiagent system

<table>
<thead>
<tr>
<th>Number of Agents</th>
<th>Jason Execute Time (magnitude)</th>
<th>eJason Execution Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>milliseconds</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>milliseconds</td>
<td>15</td>
</tr>
<tr>
<td>1000</td>
<td>seconds</td>
<td>143</td>
</tr>
<tr>
<td>10000</td>
<td>minutes</td>
<td>1550</td>
</tr>
<tr>
<td>100000</td>
<td>not measurable</td>
<td>154415</td>
</tr>
<tr>
<td>3000000</td>
<td>not measurable</td>
<td>484371</td>
</tr>
</tbody>
</table>

**Table 2.** Execution times for the greetings multiagent system

The results indicate that the multiagent systems generated by eJason scale to some hundreds of thousands of agents with an average execution time of a few seconds. Regarding the multiagents systems generated by Java-based Jason, we can see that they required more time to execute (the exact time quantities could not be precisely measured) and that it was not possible to increase the number of agents over a few thousands (in the cases labeled as *not measurable* a java.lang.OutOfMemoryError exception was raised).

Clearly these are only preliminary findings as more thorough benchmarking is needed.

## 5 Conclusions and Future Work

In this paper we have described a prototype implementation of eJason, an implementation of Jason, an agent-oriented programming language, in the Erlang concurrent functional programming language. The implementation was rather straightforward due to the similarities of Jason and Erlang. eJason is able to generate Erlang code for a significant subset of Jason. Early results are promising, as the multiagent systems running under the Erlang runtime system can make use of the Erlang lightweight processes to compose systems of thousands of agents, where the process generation, scheduling, and communication introduce a negligible overhead. We also describe and motivate some of the implementation
decisions taken during the design and implementation phases, such as e.g. the use of the ERESYE tool during an early stage and its later replacement.

Clearly, the similarities between the capabilities of agents and the Erlang processes are many, with the exception of the support for programming rational reasoning in Jason. We believe that the existence of eJason can help attract Erlang programmers to the MAS community, by providing them a convenient and largely familiar platform in which to program rational agents, while being able to implement the rest (adapting interpreter meta behaviour, and actuators for the environment) in Erlang itself. Moreover we believe that the MAS community can benefit from having access to the efficient concurrency and distribution capabilities of Erlang, while maintaining backward compatibility with legacy code, and without the need to develop a new agent-based language.

Clearly, as eJason is still a prototype, there are numerous areas for future work and improvement. The subset of Jason implemented at the moment is quite small; it is, for example, necessary to add support for belief annotations and plan labeling. Moreover, we plan to add support in eJason for different distributed agent architectures. An essential item for near future work is the implementation of a means for agents to act on their environment. We intend to make eJason agents capable to cause changes in their environment using actions programmed either in Java or Erlang, i.e., there should be no need to rewrite the large body of existing Java code for Jason environment handling. Besides, we expect to be able to use the agent inspection mechanisms already implemented in e.g. JEdit.

Another item for future work includes prototyping extensions to Jason; we believe that eJason is a good platform on which to perform such experiments. Finally we also intend to experiment with model checking, applied on the resulting Erlang code, to verify Jason multiagent systems.

References


Conceptual Integration of Agents with WSDL and RESTful Web Services

Lars Braubach and Alexander Pokahr

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{braubach, pokahr}@informatik.uni-hamburg.de

Abstract. Agent communication has been standardized by FIPA in order to ensure interoperability of agent platforms. In practice only few deployed agent applications exist and agent technology remains a niche technology that runs its own isolated technology stack. In order to facilitate the integration of agents with well-established and used technologies the connection of agents with web services plays an important role. This problem has traditionally been tackled by creating translation elements that accept FIPA or web service requests as input and produce the opposite as output. In this paper we will show how a generic integration of web services can be achieved for agents that follow the active components approach. As these entities are already seen as service providers and consumers with explicit service interfaces the integration approach will directly make use of these services. Concretely, the presented approach aims at answering two important questions. First, how can specific functionality of an existing agent system be made available to non agent systems and users. Second, how can an agent system seamlessly integrate existing non agent functionality. The first aspect relates to the task of service publication while the latter refers to external service invocation. In this paper a generic conceptual approach for both aspects will be presented and it will be shown how on a technical level an integration with WSDL and RESTful web services can be achieved. Example applications will be used to illustrate the approach in more details.

1 Introduction

On prime objective of the FIPA standards is to ensure interoperability between different agent platforms by defining e.g. the message format and communication protocols. As these standards have been made over 10 years ago they could not foresee that in practice agent technology would not be adopted to a high degree so that interoperability between agent systems is not a key concern nowadays. In practice, also the need for interoperability between different kinds of technological systems was present for a long time and with web services a set of standards has emerged that is generally accepted and has already proven its usefulness within many industry projects.
From a developer perspective standards based interconnections are important for two main reasons (cf. Fig. 1). First, specific functionality that is needed by the software to be built could be available from another vendor as a service. Hence, it would be beneficial if it is possible to seamlessly integrate such existing functionality in the agent system and hence let it reuse this outbound knowledge. Second, it should be possible to expose functionality of a newly built application in a standardized way, such that it can be easily incorporated in other external applications. Both aspects, accessing external functionality and exposing functionality to external applications, call for openness, i.e. open and standardized interfaces for encapsulating the accessed or exposed functionality in programming language and middleware independent way.

This paper tackles the question, how existing web service standards and models can be integrated into agent platforms, or more specifically:

- How can (partial) functionality be exposed as web service to external system users on demand (cf. Figure 1, left)?
- How can existing web services be integrated as functionality inside of the system in an agent typical way (cf. Figure 1, bottom)?

The rest of this paper is structured as follows. Next, Section 2 gives a background of the active components programming model. The web service integration concept is described in Section 3. Illustrative examples in Section 4 show how the concept is put into practice. Related work is discussed in Section 5, before the paper closes with conclusions and an outlook in Section 6.

2 Active Components Fundamentals

The service component architecture (SCA) is a recent standard, proposed by several major industry vendors including IBM, Oracle and TIBCO, that aims at
Fig. 2. Active component structure

providing a high-level design approach for distributed systems [7]. SCA fosters clearly structured and hierarchically decomposed systems by leveraging service orientation with component concepts. In an SCA design a distributed system is seen as set of interacting service providers and consumers that may reside on possibly different network nodes. Each component may act as service provider and consumer at the same time and defines its interfaces in terms of provided and required services in line with the traditional component definition of [13]. From a conceptual point of view SCA simplifies the construction of distributed systems but also has inherent limitations that delay widespread adoption.

Two aspects are especially crucial. First, even though a system is seen as set of interacting components, these components are rather statically connected with so-called wires between required and provided services. The underlying assumption is that at deployment time all component instances are known and can be directly bound together. This assumption is not true for many systems with components or devices appearing or vanishing at runtime. Second, the interactions between components are supposed to be synchronous. This keeps the programming model simple but may lead to concurrency problems in case component services are accessed by multiple service requesters at the same time. Low level mechanisms like semaphores or monitors have to be employed to protect the component state from inconsistencies, but these mechanisms incur the danger of deadlocks if used not properly.

Active components build on SCA and remedy these limitations by introducing multi-agent concepts [3,10,11]. The general idea consists in using the actor model as fundamental world model for SCA components [6]. This means that components are independently behaving actors that do not share state and communicate only asynchronously. In this way concurrency management can be embedded into the infrastructure freeing developers from taking care of ensuring state consistency of components.¹ As it is fundamental property of the actor

¹ Asynchronous communication helps avoiding technical deadlocks. Of course, at the application layer circular waits can be created making the components wait on each other. In contrast, to a technical deadlock, in this case no system resources (threads) are bound to the waiting entities. Furthermore, the “application deadlocked” components remain responsive and can answer other incoming service requests or act proactively.
model that actors come and go over time, active components do not encourage static binding of components but instead rely on dynamic service discovery and usage. In Fig. 2 an overview of the synthesis of SCA and agents to active components is shown. It can be seen that the outer structure of SCA components is kept the same and the main difference is the newly introduced internal architecture of components. The internal architecture of active components is used to encode the proactive behavior part of a component that can be specified additionally to provided service implementations. More details about the integration can be found in [3].

3 Web Service Integration Concept

In this section the overall integration concept as well as more detailed design choices will be presented.

3.1 Making Functionality Accessible as Web Service

In order to make functionality of an application available to external system users, component services can be dynamically published at the runtime of the system. In general, it has to be determined when, where and how a service should be published. A common use case consists in performing the service publication according to the lifecycle state of the underlying component service. For this reason, per default, service publication and shutdown is automatically checked for when a component service is started or ended. The component inspects the provided service type descriptions for publish information and indirectly uses this information to publish/shutdown the service via delegation to the infrastructure. The publish information is composed of four aspects: publish type, publish id, mapping and properties. To support arbitrary kinds of service publications such as REST (Representational State Transfer) [4] and WSDL (Web Services Description Language) [15] the component uses the publish type from the component specification of a service to dynamically search for a suitable service publisher (cf. Figure 3). Subsequently, the component asks the service publishers one by one if it supports the requested publish type. In case a suitable publisher could be found it is instructed to publish or retreat the component service. Otherwise the publication has failed and an exception is raised within
the service provider component. As an additional task the publisher may also support the advertisement of the newly deployed service within a service registry that can be accessed from external users. The service user can use the service information from the registry to locate the service and issue requests to it. The service itself acts as a proxy, which forwards the request the actual service provider and waits for a reply. The service provider executes the service domain logic and returns the service results to the proxy which in turn delivers it to the external user. It has to be noted that the incoming web service request is synchronous and therefore blocks until the internal asynchronous component service processing has been finished. In the following it will be shortly explained how WSDL and REST publishing work.

**WSDL Publisher** Conceptually, a direct correspondence between methods of the component service and the operations of the WSDL service is assumed, i.e. both services are syntactically and semantically equivalent with one minor exception. The exception is that WSDL services are mostly assumed to be synchronous whereas component services follow the actor model and are asynchronous. Therefore, publishing a component service requires the original asynchronous service interface being rewritten as synchronous version. Based on this interface the proxy web service component can be automatically generated using dynamic class creation using bytecode engineering or Java dynamic proxies.

Publishing a WSDL service is supported extensively in Java environments and also directly within the JDK. Most web containers like Axis2 and JDK internal lightweight container allow publishing annotated Java pojos (plain old java objects, i.e. simple objects). The container automatically reads the Java interface of the pojo and uses the additional annotation information to produce a WSDL description of the service. Java types are mapped using JAXB\(^2\) databinding to corresponding or newly created XML schema types. In normal cases the message signatures of the Java interface are sufficient for creating the WSDL and only for edge cases further annotation metadata needs to be stated. For Jadex, therefore a default WSDL publisher is provided that creates an annotated Java pojo based on the supplied synchronous service interface and feeds this into the web container. The container makes available the new service under the given URL.

**REST Publisher** REST service interfaces are potentially very different from object oriented service interfaces as they follow the resource oriented architecture style \[4\]. In REST the idea is that services work with resources on web servers and employ the existing HTTP communication protocol to address these resources via URIs. In addition, REST proposes special semantics to the different kinds of HTTP requests, e.g. a GET request should be used to retrieve a resource and PUT to create a new one. Taking this into account, a one-to-one mapping between method signatures of the object oriented service interface and

\(^2\) [http://jaxb.java.net/](http://jaxb.java.net/)
REST methods is not directly possible or the ideal result.\(^3\) Hence, the idea is to allow a very flexible mapping between both kinds of representations. In general, three different types of mappings are supported ranging from fully automatic, over semi automatic with additional mapping information to completely manual descriptions. Mapping information that needs to be generated encompasses the set of methods that should be published and for each method the following information:

- **URL**: i.e. the address that can be used to reach the service method. Typically, the URL of a service method is composed of two sections. The first section refers to the service itself and the second section refers to the method. This scheme treats methods as subresources of the service resource. In case multiple methods with the same name but different signatures exist it has to be ensured that different URLs are produced.

- **Consumed and produced media types**: REST services are intended to be usable from different clients such as browsers or other applications. These clients may produce and consume different media types such as plain text, XML or JSON. The REST service can be made accepting and producing different media types without changing the service logic by using data converters like JAXB for XML and Jackson\(^4\) for JSON. The conversions from and to the transfer formats are done automatically via the REST container infrastructure respecting the given media types.

- **Parameter types**: i.e. the parameter types the rest service expects and the return value type it produces. In the simplest case these types directly correspond to the object oriented parameter types of the underlying service interface but often RESTful APIs intend to use basic string parameters in the URL encoded format of HTTP. If there is a mismatch between the object oriented and the RESTful interface, parameter mappers can be employed that transparently mask the conversion process. It has to be noted that the transformation of parameter values is n:m, meaning that n input values of the component service need to be mapped to m parameters of the REST service. Therefore, it has to be ensured that as well more than less parameters can be generated from the incoming value set. Parameter type generation is done in addition to conversions with regard to the consumed and produced media types.

- **HTTP method types**: REST defines specific meanings for HTTP method types like put, get, post, delete that roughly correspond to the CRUD (create,

\(^3\) It has to be noted that characteristics like stateless interactions and cacheability, which are often associated with REST services, do not render REST useless for implementing multi-agent interactions. First, the web resources in REST are stateful being subject of creation, manipulation and deletion. Second, cacheability means that operations should be idempotent, which is achieved when e.g. mapping parts of an interaction protocol to the HTTP request types GET and HEAD. Given that for each stateful interaction a new REST resource is created both properties can be preserved.

\(^4\) [http://jackson.codehaus.org/](http://jackson.codehaus.org/)
retrieve, update, delete) pattern. This means that different HTTP method types should be used depending on the action that should be executed on a web resource. Mapping these types from a method signature is hardly possible as the method semantics is not available to the mapper. Nevertheless, using other HTTP methods than originally intended is not prohibited per se. Possible negative effects that may arise concern efficiency as some of the method types are considered being idempotent so that existing HTTP caching can further be used.

The architecture of the REST publisher is more complex than the WSDL publisher. It partitions work into two phases. In the first phase the given component service interface is analyzed with respect to the methods that should be generated and how these should be represented in REST (according to the descriptions above). The result is a list of methods with exact descriptions how these methods should be created in REST. This list is passed on to the second stage in which a Java class is generated for the REST service via bytecode engineering. The generator first creates Java method signatures using the method name and parameter types produced in the first phase. Afterwards, it creates Java annotations for the REST specific mapping information according to the JAX-RS specification. The publisher can directly pass this class to the REST container, which ensures that the service is made available.

3.2 Integrating Existing Web Services

Integrating web services aims at making usable existing functionalities as component services (cf. Fig. 4). In this way access to external functionalities can be masked and be used in the same way as other middleware services. Challenges in this integration are mainly limited to the question how an external service can be adequately mapped to the middleware and how it can be made accessible to service clients. The conceptual approach chosen is based on wrapper components, which act as internal service providers for the external functionality. A wrapper component offers the external functionality as provided service with

Fig. 4. Web Service Invocation
an interface that on the one hand mimics the original service interface and on
the other hand complies to the asynchronous requirements of the middleware,
i.e. in the simplest case the internal interface is the asynchronous version of the
external interface. The implementation of the provided service is represented
by a specific forward mechanism that dispatches the call to the external web
service. To resolve the synchronous/asynchronous mismatch a decoupled invo-
cation component is used. For each service call such an invocation component is
created, which is solely responsible to perform the synchronous operation. While
the operation is pending the invocation component remains blocked but as it has
no other duties than performing the call this is not troublesome. This pattern
keeps the wrapper component responsive and lets it accepting concurrent service
invocations without having to wait until the previous call has returned.

WSDL Wrapper The WSDL wrapper component heavily relies on the exist-
ing JAX-WS technology stack. One core element of this stack is a tool called
"wsimport" that is used to automatically generate Java data and service classes
for a given WSDL URL. The generated code can directly be used to invoke the
web service from Java. Based on this generated code the asynchronous service
interface has to be manually defined relying on the generated data types for
parameters. For this reason no further parameter mappings need to be defined.
The wrapper component itself declares a provided service with this interface and
uses a framework call to dynamically create the service implementation.

REST Wrapper The REST wrapper is based on JAX-RS technology but cur-
rently does not employ automatic code generation. Instead, the asynchronous
component service interface has to be created manually based on the REST
service documentation. The interface definition should abstract away from the
REST resource architecture and give it a normal object oriented view. The map-
ping of the component service towards the REST service is done with a mapping
file represented as annotated interface. This annotated interface contains
all methods of the original service interface and adds mapping information for
the same types of information that already have been used for publishing, i.e.
for each method the URL for the REST call, the consumed and produced media
types, parameter and result mappings as well as the HTTP method type. The
wrapper component definition is done analogously to the WSDL version with
one difference. Instead of using a generated service implementation, the REST
wrapper uses a dynamic proxy that uses the mapping interface to create suitable
REST invocations.

6 http://jax-ws.java.net/
7 Automatic code generation can only be used for REST services that supply a web
application description (WADL file) of themselves that represents the pendant to the
WSDL file of an XML web service. Similar to wsimport, a tool called wadl2java is
available that is able to create Java classes for data types and services of the REST
service. A problem is that WADL has not reached W3C standard status and also is
not in widespread use in practice.
public interface IBankingService {
  public IFuture<AccountStatement> getAccountStatement(Date begin, Date end);
}

public interface IWSBankingService {
  public AccountStatement getAccountStatement(Date begin, Date end);
}

@Agent
@ProvidedServices(@ProvidedService(type=IBankingService.class,
  implementation=@Implementation($component)
  publish=@Publish(publishType="ws", publishId="http://localhost:8080/banking",
  mapping=IWSBankingService.class)
  public class BankingAgent implements IBankingService {
    ...
  }
}

Fig. 5. Java code for publishing a WSDL service

4 Example Applications

In this section the publish and invocation web service integration concept will be further explained by using small example applications. The domain used to show how service publication can be achieved is a simple banking service, which offers operations for account management. For simplicity reasons it has been stripped down to one method called getAccountStatement, which is used to fetch an account statement viable for a specifiable date range. Integration of external services is shown using a WSDL geolocation service for IP addresses and the Google REST chart API.

4.1 WSDL Publishing

Figure 5 shows how the service publication is specified in the Jadex active components framework. The existing component service interface IBankingService (lines 1-3), which uses asynchronous future [12] return values\(^8\) (see line 2) is augmented with a synchronous interface IWSBankingService (lines 5-7) providing the same methods. In the component definition (lines 9-16) the declaration of the provided service (lines 10-13) is extended with the publish information (lines 12, 13) specifying the target URL and the newly defined synchronous interface (line 12). In the publish information the publish type is set to WSDL web services (ws). In this example, the banking service is implemented by the component itself (line 14), which is stated in the provided service declaration using the predefined variable $component (similar to this in Java).

\(^8\) A future represents the result of an asynchronous computation, i.e. the future object is immediately returned to the caller will contain the real result value when it has been computed. The caller can use the future to check if the result already has been produced or use a listener to get a notification when this happens.
public interface IBankingService {
    public IFuture<AccountStatement> getAccountStatement(Date begin, Date end);
}

@Agent
@ProvidedServices(ProvidedService(type=IBankingService.class,
    implementation=Implementation($component)
    publish=Publish(publishType="rs", publishId="http://localhost:8080/banking")
    public class BankingAgent implements IBankingService {
    ...
    }

Fig. 6. Java code for publishing a REST service

public interface IRSBankingService {
    @GET
    @Path("getAS/")
    @Produces(MediaType.TEXT_HTML)
    @MethodMapper(value="getAccountStatement", params={Date.class, Date.class})
    @ParametersMapper(@Value(clazz=RequestMapper.class))
    @ResultMapper(@Value(clazz=BeanToHTMLMapper.class))
    public String getAccountStatement(Request request);
}

Fig. 7. REST publish mapping information

4.2 REST Publishing

As introduced earlier, REST publishing is supported in fully automatic, semi automatic and manual modes. In Figure 6 the fully automatic variant is shown, which is similar to the WSDL variant but doesn’t require a synchronous interface to be manually derived. In contrast to the example above, the publish type is set to REST services (rs, line 8). The fully automatic mode uses internal heuristics to generate appropriate REST methods, which is difficult in many cases. Hence, additional mapping information can be supplied in both other modes. For this purpose an annotated Java interface or (abstract) class can be employed.

In case of an interface the method signatures are enhanced with REST annotations as shown in Figure 7. It can be seen that a method getAccountStatement() with one parameter of type Request (line 8) is delegated to a component service method with the same name but other parameter types. The method mapper annotation is used to specify the target method (line 5) and additional parameter and result mapper can be added to transform the corresponding values (lines 6 and 7). In this case a request mapper is used to extract two dates from a request and the result is generated as HTML using a simple bean property mapper. This example also shows the difference between parameter and media types. The Java return type in this example is string but the additional produces annotation (line 4) tells the client that it can expect HTML.

If even more flexibility is needed, instead of an interface a class can be used (not shown). In this class it is possible to add abstract methods and anno-
tate them in the same way as in the interface. Additionally, other non abstract methods can be implemented with arbitrary domain logic to bring about service functionalities. If no generation is wanted, the wrapper class can also be implemented completely by the programmer.

In Fig. 8 a screenshot of the banking REST web interface is shown. This web site is produced by a banking agent with a publish annotation as shown above. This interface is automatically generated by the `getServiceInfo()` method and is per default linked to the root resource URL of the service (here localhost:8080/banking1/). It can be seen that the web site contains a new part for each service method with basic information about it, i.e. the method signature, REST call details, the URL and a form with input fields for all parameters. According to the media types the service method is able to consume a choice box is added to allow the user specifying in which format the input string shall be interpreted. This can be seen in the `getAccountStatement()` method, which accepts JSON and XML. Currently, the result value of a method call is produced in the same media type as the request but it is easily possible to add another control that allows to request the service to produce an alternative format.

4.3 WSDL Invocation

WSDL service invocation is illustrated using a geo IP service, which offers a method to determine the position of an IP address. After having generated the Java classes for data types and service using `wsimport`, based on the generated service interface an asynchronous version needs to be defined (cf. lines 1-3 in

**Fig. 8. Banking REST web service screenshot**

```java
IBankingService

getAccountStatement(Date, Date)
POST Consumes application/vnd.api+json

getServiceInfo()
GET Produces application/vnd.api+json

http://localhost:8080/banking1
http://localhost:8080/banking1
```
public interface IGeoIPService {
    public IFuture<GeoIP> getGeoIP(String ip);
}

@Agent
@ProvidedServices(
    @ProvidedService(type=IGeoIPService.class,
    implementation=Implementation($component.createServiceImplementation(
    new Mapping(GeoIPService.class)))))
public class GeoIPAgent {
    ...
}

Fig. 9. Java code for invoking a WSDL service

Figure 9). In the component declaration (lines 4-8) a provided service is specified using
the asynchronous service interface (line 5) and an automatically generated
implementation (lines 6-7). The framework method that is called to create the
implementation takes as argument the wsimport generated service class.

4.4 REST Invocation

REST invocation is exemplified using the Google chart API, which can be used
to create chart images of different types for a given data set. The implementation
is shown in Figure 10. It consists of the asynchronous service interface (lines 1-
4), the REST service mapping (lines 6-13) and the chart component definition
(lines 17-23). For illustration purposes the component interface is reduced to one
method that can be used to create a bar chart. The method expects the width
and height of the image to produce, possibly multiple data series, label texts
and series colors as input and produces an png image as output. The mapping is
defined within an interface called IRSChartService (lines 6-15). It declares that
the generated REST call uses HTTP GET on the google chart URL. In addition,
parameter mappers for in- and output values need to be employed (lines 10-11,
mapper code not shown) as the REST API expects a specific textual encodings
for the data. The component implementation is very similar to the WSDL variant
with exception of the mapping definition in terms of an interface.

Fig. 11 shows a chart application screenshot. In the background the Jadex
control center window of the platform is displayed while in the foreground the
chart window is shown. The application consists of two agents. The Chart-
Provider agent that takes over the wrapper role and offers an IChartService in-
stance and on the other hand the ChartUser agent which own the graphical user
interface for entering chart requests and displaying the resulting chart graphics.
On the lower left hand side of the control center the running agent instances
with required and provided services are depicted. It can be seen that the Chart-
Provider offers an IChartService and the ChartUser requires an IChartService.
The processing is done as follows. After a user has entered some configuration
data in the chart window including e.g. width and height of the target image,
series data, and colors, and issued a chart request via pressing the draw but-
public interface IChartService {
    public IFuture<Byte[]> getBarChart(int width, int height, double[][] data,
    String[] labels, Color[] colors);
}

public interface IRSChartService {
    @GET
    @Path("https://chart.googleapis.com/chart/")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    @ParametersMapper(@Value(clazz=ChartParameterMapper.class))
    @ResultMapper(@Value(clazz=ChartResultMapper.class))
    public IFuture<Byte[]> getBarChart(int width, int height, double[][] data,
    String[] labels, Color[] colors);
}

@Agent
@ProvidedServices(@ProvidedService(type=IChartService.class,
    implementation=\$component.createServiceImplementation(
    IRSChartService.class))
public class ChartAgent {
    ...
}

Fig. 10. Java code for invoking a REST service
	on, the chart user agent fetches its required chart service (which is dynamically searched on request) and calls the \texttt{getBarChart()} method. The service call is received by the user agent, which automatically transfers it to a REST call and hands it over to the external REST provider. The result is passed back to the user agent which displays the corresponding chart in the window for the user.

5 Related Work

In this section, the features of the approach proposed in this paper will be discussed with respect to the following areas of related work: 1) \textit{programming level frameworks}, i.e. APIs and tools that ease the usage of web services from inside a general purpose programming language like Java, 2) \textit{middleware extensions} that aim at a conceptual integration between web services and agent middleware and 3) \textit{SCA standards and implementations} that, although they don’t focus on asynchronous programming, are an important conceptual inspiration of this work.

The approach presented in this paper is unique with respect to the simultaneous conceptual treatment of both directions of web service integration: publication and access. Treating both the same way has advantages e.g. with regard to developers only having to learn one API for both aspects. Programming level frameworks such as JAX-WS and Axis² also follow this direction to the ad-

² http://axis.apache.org/axis2/java/core/
vantage of the programmer. E.g. in JAX-WS the developer can use the same
techniques to generate Java classes and interfaces from an existing WSDL or
vice versa, regardless if she wants to publish or access a web service. Interest-
ingly, the conceptual integrations of middleware extensions focus usually on only
one aspect. E.g. in the area of agent platforms, [5,9,2] are examples for dealing
with exposing agent services as web services. On the contrary [14,8] discuss web
service invocation from agents. The ProActive middleware [1] provides support
both for web service invocation as well as web service publication. Yet, only the
publication part provides a conceptual integration into the ProActive program-
ing model by allowing to directly expose methods of ProActive objects as web
services. The invocation part on the other hand is merely a set of utility classes
comparable to other programming level frameworks. Unlike the aforementioned
approaches, the SCA standards treat service publication and invocation at the
same conceptual level. Due to the prevalent synchronous programming model,
SCA lacks an additional wrapper level for decoupling caller and callee during
service invocation or execution.

Another important aspect of the approach presented here is the unified treat-
ment of WSDL and RESTful web services. Most existing integration work is
devoted to WSDL web services, e.g. [5,9,8,14] in the agent area and also imple-
mented in ProActive. The main reason for this is probably the explicitly typed
nature of the WSDL that lends itself to automatic code generation. REST on
the other hand is much more free in the way a service is defined and used and
thus requires more manual implementation or mapping specification. Publica-
tion of REST services is treated in [2], although they only support a simplistic
mapping of only one operation per service. Similar to the conceptual middle-
ware extensions, most programming level frameworks focus on one type of web
service, with many standards (JAX-WS and JAX-RS) and non-standards based
implementations being available for each type. One exception is Apache CXF\textsuperscript{10}, that incorporates APIs for RESTful as well as WSDL services. Yet, CXF does not aim at unification for the programmer, but at implementing the different available standards. The SCA standards only deal with WSDL web services and define a ws binding for provided and required SCA component services. Some available SCA implementation like Tuscany\textsuperscript{11} and Frascati\textsuperscript{12} additionally offer proprietary support for RESTful services. Yet, both require JAX-RS annotations in the service implementations that hinder a transparent usage of the same component functionality as WSDL and REST service.

In summary, the approach presented in this paper picks up earlier work on web services support for agent platforms, incorporates and extends existing ideas from SCA and combines a unified treatment of REST and WSDL with a conceptual model for an agent-style asynchronous provision and invocation of services.

6 Conclusions and Outlook

Web services are important for interoperability and extensibility as they allow integrating external functionality into applications as well as developed functionality being integrated in external applications. This paper focuses on web services support for agent platforms.

The proposed model provides a conceptual integration for both the publication of application functionality as web service as well as the invocation of external web services. To avoid dependencies between the implementation of application functionality and specific web services technology such as WSDL or REST, the model incorporates two important abstraction layers. First, the wrapper and invocation agents map a synchronous external web service interface to an asynchronous one and register the mapped service description transparently inside the middleware, such that external and internal services can be access in the same way. Second, publish services take care of exposing internal services as external web services and different publish services for REST and WSDL technology allow the same internal services to be transparently published using these different approaches.

The integration concept has been implemented as part of the open source active components platform Jadex\textsuperscript{13}. Besides the simple examples presented in this paper, the web services integration is currently being put into practice in a commercial setting that deals with business intelligence processes and activities in heterogeneous company networks.

\textsuperscript{10} http://cxf.apache.org/
\textsuperscript{11} http://tuscany.apache.org/
\textsuperscript{12} http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti
\textsuperscript{13} http://jadex.sourceforge.net/
References


Agent Programming Languages Requirements
for Programming Cognitive Robots

Pouyan Ziafati¹,³, Mehdi Dastani³, John-Jules Meyer³, and Leendert van der Torre¹,²

¹ SnT, University of Luxembourg
² CSC, University of Luxembourg
³ Intelligent Systems Group, Utrecht University
{pouyan.ziafati,leon.vandertorre}@uni.lu
{M.M.Dastani,J.J.C.Meyer}@uu.nl

Abstract. This paper presents various requirements for agent programming languages to provide better support for implementing autonomous robotic control systems. One of such requirements is providing built-in support for integration with existing robotic frameworks such as Robot Operating System (ROS). Another requirement is real-time reactivity to events. Real-time reactivity of BDI architecture is discussed and distributed BDI architecture is proposed as a promising solution for addressing real-time concerns. Another requirement is extending the BDI architecture with a sensory component to be used for the management of heterogeneous sensory data and reasoning on high-level events. The last requirement is the representation of complex plans and coordination of the parallel execution of plans. A checklist of the required plan execution control mechanisms based on an analysis of plan representation and execution capabilities of various robotic plan execution languages such as PDL, APEX, PLEXIL and PRS is provided.

Keywords: Agent Programming Languages, Cognitive Robotics

1 Introduction

Today’s service robots have emerging applications in domestic, military, healthcare and entertainment domains. These applications demand ever increasing levels of intelligence and autonomy constituting the typical challenge for the field of cognitive robotics. This is a branch of robotics aiming at studying and developing robots with reasoning capabilities needed to achieve complex goals in dynamic environments. An ability to reason about objectives to select appropriate actions empowers a robot with the so-called deliberative behavior.

One of the most suitable architectures for implementing deliberative behavior is the BDI architecture [28, 29] inspired by the BDI (Belief-desire-intention) model of human practical reasoning [22]. This architecture includes components such as beliefs, goals, plans, and plan generating rules. Each plan generating rule specifies a plan reaching a goal if executed in a specific belief state. The
deliberative behavior in BDI architecture is a cyclic process in which sensory information is processed, beliefs and goals are updated, applicable plan generating rules are selected, and applied, and then generated plans are executed.

Various agent programming languages have been designed and developed to facilitate the implementation of BDI architecture. Examples of these programming languages include 2APL [9], AgentSpeak(L) [1], Jason [4], Jack [36] and MetateM [12] (see [3] for a survey of agent programming languages). However, the application domains of these languages have been mainly limited to cognitive software agents. One reason for this might be due to their lack of necessary support for addressing different requirements of robotic control systems. This urges system developers to invent many ad hoc solutions for such requirements, making development of such systems costly and hard to maintain [21, 31]. Moreover, the focus of robotic research during the last decades has been mostly on low-level robotic functionalities such as SLAM, low-level control, face recognition, voice recognition, etc. However, recent achievements of robotic research in providing advanced perception and control functionalities, and utilizing software engineering techniques for managing complex robotic software provide a necessary basis for developing robotic systems with high levels of intelligence and autonomy.

Our research aim is to provide necessary methodologies and requirements to facilitate the use of BDI-based agent programming languages for implementing robotic control systems in a modular and systematic way. This paper contributes in bringing together the research on design and development of cognitive agents, focusing on specification of agents based on their mental attitudes and cognitive processes, and research on design and development of real robotic systems. In particular, problems of plan execution control, sensory data management and real-time reactivity for robotic systems in single robot applications are discussed. Considering these problems, various requirements are identified that agent programming languages should address to be more suitable for programming cognitive robotic systems. In order to identify these requirements, we ground our discussion on the robotic literature containing much different research work on developing various robotic systems, addressing the above problems for different application domains. Our domain of interest is cognitive robotics normally demanding a higher level of autonomy and intelligence. For this reason, we generalize from the analyses of different robotic systems.

The remainder of this paper is organized as follows. Section 2 presents an application scenario that is used throughout the paper to provide illustrative examples for our discussions. Section 3 briefly discusses the advantages of robotic frameworks regarding low-level sensory data processing and action control and, furthermore, presents ROS open-source robotic framework. Section 4 is devoted to a short discussion of balancing between proactivity and reactivity and how such requirement is addressed by different robotic architectures. Real-time properties of the BDI architecture are investigated and based on ideas from state of the art robotic architectures; research toward developing distributed BDI architectures is proposed as a promising research direction for developing real-time BDI-based control systems. Section 5 discusses the need for an extension of the
BDI architecture with a sensory component to be used for the management of heterogeneous sensory data and reasoning on high-level events, and provides a checklist of requirements for such a component. Section 6 surveys the plan representation and execution capabilities of existing robotic systems and presents such capabilities as a checklist of different requirements necessary for synchronizing the execution of actions of plans and for coordinating the parallel executions of plans. Section 7 concludes the paper.

2 An Application Scenario

Araz and Mori are living in a shared room. They are both old and have Alzheimer. Moreover Mori is under medication. To help them living easier and increase their safety, their children have bought them a NAO robot personal assistant. NAO helps them by performing the following tasks:

1. T1: To remind Mori to take drug A every morning at 10 am. To remind Mori, NAO calls: “Mori, take A.”. When NAO hears the response back: “OK, I take A.”, it considers the task as finished successfully.

2. T2: To check if drug A is finished, and then call the drugstore to order. Drug A’s color is red and is placed in a white box. NAO should check the box every afternoon and if there were no more A in the box, it should press a BLUE BUTTON provided in the room, which causes ordering the drug automatically.

3. T3: To open the door if a visitor rings the bell. NAO checks the visitor face from the door camera; if it recognizes the face, it opens the door by pressing the OPEN DOOR BUTTON. Otherwise it informs Araz and Mori by calling: “A stranger is behind the door.”. In this case, the task is finished when NAO hears the response back: “Ok, I check it.”.

4. T4: To frequently check if there is any trash (i.e. a black cube) on the table and remove it to the trash can.

5. T5: To remind Araz and Mori about the places of their personal objects in the following way: Mori/Araz asks NAO: “It’s Mori/Araz.”, “Remember my key is on the desk.” and later can ask NAO: “It’s Mori/Araz.”, “Where is my key?” then NAO should answer “On the desk.”.

6. T6: To wake them up each morning by playing a piece of music at a specific time depending on the day of a week.

7. T7: If Araz or Mori calls: “Help!”, NAO should call the doctor as soon as possible by pressing a RED BUTTON placed in the the room. Also in T1 and T3, If NAO communicate with Araz/Mori for 3 times and does not hear the response back, NAO calls the doctor as it might be sign of a dangerous situation.

2.1 Constraints

Some of the constraints and priorities applied in performing above tasks are as follows:
– When NAO wants to press a button, its hand should be empty. For example, NAO cannot open the door or call the doctor, when carrying trash.
– NAO cannot be at more than one location at the same time, so when it has two different goals to go to two different locations, these goals are conflicting.
– To perform T2, NAO should first go to the location L and stop there in such a way that its orientation is O to be in a certain distance from the drug box and maintain its camera’s orientation towards the box. Then NAO should keep this location and orientation while taking pictures to analyze if the box is empty.
– When parallel execution of plans have a conflict, priorities of the tasks from the highest to the lowest are: T7, T5, T3, T2, T1, T6, T4.
– There are two classes of priorities: Non-preemptive priorities (T1 to T6) which are only applied when choosing which plan to start first, and preemptive priorities (T7) which are applied at any time and can enforce the executions of other plans to be stopped if necessary. For example if the door rings when NAO has already picked up a trash, NAO first finishes putting the trash in the trash can and then opens the door, but when Araz/Mori calls for help, NAO leaves any other tasks and in this case throws the trash away instantly and performs T7.

3 Integration with Robotic Frameworks

Current robotic control architectures agree on the need for a functional layer to provide different robot’s action and perception capabilities and processing algorithms such as image processing, path planning and motion control in a modular way. Robotic control architectures differ in the way functional layer modules are controlled to achieve more complex tasks. A functional layer is usually composed of a large number of modules. In order to cope with ever growing scale and scope of robotic systems, a wide variety of robotic frameworks has been developed to facilitate the development, re-use and maintainability of the functional layer modules [20, 17]. Advantages of these frameworks include: providing standard interfaces for accessing heterogeneous robotic hardwares; facilitating robotic software development and reuse by using software engineering techniques such as component based software development; providing software development tools such as programming environments; and providing open-source repositories of robotic softwares.

**ROS** An example of such robotic frameworks is ROS (Robot Operating System)[32] which has currently become the de-facto standard open source robotic framework. The ROS repository has an ever increasing number of state-of-the-art software packages for interfacing various robotic hardware, and for performing different robotic tasks such as SLAM, image processing, etc. Providing access to such advanced robotic software packages significantly eases the rapid prototyping and development of complex robotic applications. ROS is a highly flexible framework for developing a robotic functional layer. Using ROS, functional
layer modules (i.e. nodes) can be developed in different languages such as C++, Python and Java. These modules can be started, killed, and restarted at run-time and communicate with each other in a peer-to-peer fashion. Several different styles of communication between modules are provided, including synchronous service based (i.e. request/reply) interaction, asynchronous publish-subscriber based streaming of data, and key-value based storage/retrieval of data on/from a central server. ROS modules communicate by exchanging messages based on a simple standard language similar to C language data structures. Ros supports robotic simulators such as Stage, Gazebo and MORSE. Moreover ROS has been integrated with many other robotic frameworks such as OpenRAVE, Orocos, and Player.

Similar to the others, a BDI-based robotic control system also needs a functional layer to interface with robotic hardware and provide sensory data and actions in different levels of abstraction. To facilitate the use of agent programming languages for developing robotic control systems, such languages should provide suitable interfaces to integrate with existing robotic frameworks. Such interfaces ideally provides built-in support for communication and control mechanisms of robotic frameworks.

4 Real-time Reactivity

Providing a proper balance between deliberation and reaction has been always a major concern in research on robotic control systems. On the one hand, a complex deliberation capability is desired for an autonomous robot to generate plans to achieve its goals, taking into account its limited resources, and on the other hand, it requires a time-bounded reactivity to events it receives from dynamic environments. Time-bounded reactivity to events is essential for an autonomous robot for its safety, the safety of its environment and for different robot’s functionalities.

Early examples of robot control architectures such as Shakey [24] were based on the SPA (i.e. sense-plan-act) paradigm. The SPA architecture embeds the deliberative component of a robot (i.e. a component responsible for time consuming decision-making tasks such as classical AI planning) at the heart of the robot’s control loop. The problem is that in dynamic environments a generated plan might become invalid before the plan can be fully executed. Moreover while a robot is deliberating, it is unable to react to events.

An alternative for designing a robot’s control architecture is behavior-based robotics (i.e. reactive). Pioneered by Brooks’ subsumption architecture [5], the behavior-based robotics aims to drive a robot’s control behavior without explicit representation of the robot’s world model. In this paradigm, rather than having a planning capability or an explicit goal-oriented behavior, a robot’s behavior is emerged as the result of the behavior of the robot’s reactive components, each usually based on simple stimulus-response rules. Although behavior-based robotics has shown to be successful in many applications, it has been argued that such an approach to modeling intelligence is incapable of scaling up to
human-like intelligent behavior and performance which is the motivation of this research [35, 8].

**Three Layered Architecture** To address the above issues, providing deliberation capability and at the same time preserving reactivity, robotic research has come up with different hybrid architectures. Perhaps the most well known and used hybrid architecture is the classic three layered architecture [14, 18]. This architecture composed of 3 components. A functional component interfacing with hardware and providing low-level perception and action capabilities. A deliberation component producing plans to achieve a robot’s goals and supervising the temporal execution of plans. Finally, an executive (i.e. sequencer) residing between the other two and its main functionality is context-dependent execution of plans generated by the deliberation component. The executive refines plans into low-level actions, which can be executed to control modules of the functional component. It reacts also to events and provides limited monitoring and plan failure recovery mechanisms.

The design principle behind the three layered architecture is to encapsulate the time-consuming deliberation processes into a deliberation component and increase reactivity by providing a separate executive component with a much faster reaction time than that of the deliberation component. Different procedural and task description languages (e.g. TDL [34], PRS-lite [23]) are used to develop executive systems. Executive systems are intended to operate in real-time environments; however, most of them do not provide proofs for having real time properties. One exception to this is PRS [19] guaranteeing a bounded reaction time and some other real-time properties under certain conditions. PRS is a BDI-based procedural reasoning system making its analysis a good starting point for discussing about real-time properties of the BDI architecture.

**Real-time Properties of a BDI Architecture** Real-time systems should guarantee bounded reaction and response time to events. Similar to the work of Ingrand and Coutance [19], we consider the reaction time as the delay between the time point at which an event is received by the system and the time point when it is taken into account by the system (i.e. read from the input buffer), and the response time as the delay between the time point at which an event is received by the system and the time point when the generated plan for that event has been completed.

PRS has a BDI-based deliberation cycle in which: 1) new events and internal goals received during the last deliberation cycle are considered; 2) based on these new events, goals, and system beliefs, a plan which is in PRS called Knowledge Area (KA) is selected; 3) the selected KA is placed on the intention graph; 4) an intention is chosen; 5) and at the end, one step of the active KA in the selected intention is executed, which can result in a primitive action or the establishment of a new goal. In a PRS application system, in addition to domain specific KAs, there can be also a number of meta-level KAs which are used for example to implement different methods for choosing among multiple applicable KAs or
different methods to choose the current active intention to be considered for the execution. Such meta-level KAs are processed in an inner loop inside the main deliberation cycle of PRS.

PRS can be proven to have a bounded reaction time if certain conditions are met [19]. In each deliberation cycle, PRS executes only a single step of a single intention. This is not an efficient strategy in many cases as execution of each deliberation cycle has a minimum computational cost (regardless of the number of new events and goals) which can result in starvation of intentions. Moreover, in each PRS deliberation cycle, only a single KA can be selected to be intended. However, in order to generate different plans to respond to different events and to achieve different goals, normally more than a single KA is needed to be intended. It can be shown that intending only a single KA and executing only a single step per deliberation cycle are not necessary requirements for PRS to have a bounded reaction time. Only the maximum number of steps that can be executed and the maximum number of new goals that can be established in each deliberation cycle should be bounded. However, intending more than a single KA and executing more than a single step in each deliberation cycle increases the upper bound on the reaction time of the system.

A BDI-based system served as an executive component of a three-layered robot control architecture is meant to provide lighter functionalities than a BDI-based system encoding the entire logics of a robot control system. In the former case, it is mostly the responsibility of the deliberation component to generate high-level plans and supervise their temporal executions. For example it is mostly the case that the deliberation component reasons on priorities of goals and resource constraints to decide which goal should be followed at the moment and sends that goal to the executive to generate and execute a plan for that goal. But in the latter case, those functionalities should be encoded in the BDI-based system itself. Moreover, as is explained in section 5, and in section 6, to be suitable for implementing control systems of autonomous robots, a BDI-based control architecture should support different mechanisms for sensory data management, plan execution control and resource management. Providing such functionalities makes the evaluation of real-time properties of such BDI-based system a tedious task if not impossible, and can increase the upper bound on the system reaction time (if exists), so that the system cannot meet its required minimum reaction time.

**Toward a Real-time Distributed BDI Architecture** In many robotic applications, a robot needs to guarantee real-time properties only for a small subset of its tasks which are critical for safety reasons or the robot functionality. For example in the presented application scenario, NAO only needs to guarantee a bounded reaction time when a user asks for help. In cases where the nature of a problem allows for its decomposition into a set of tasks with different priorities and real-time constraints, distributed control architectures can provide effective solutions. In a distributed control system, a set of control components with different levels of computational complexity (i.e. deliberation capabilities) can be
utilized to provide solutions for different tasks according to their real-time requirements. For example some BDI-based control components can be devoted to implement simple event handling tasks to guarantee real-time reaction and response to critical events and other BDI-based control components can be used to implement complex goal-based deliberative behaviors with relaxed real-time properties. In addition to facilitate the development of real-time control systems, distributed control architectures are also effective in dealing with software development complexity [7] and allow for easier and more efficient use of parallel and distributed computing resources whenever available.

To facilitate the development of real-time BDI-based distributed control systems, suitable methodologies and tools are needed to develop real-time BDI-based control components and analyze and guarantee their real-time properties. Also a dedicated architecture is required to provide necessary mechanisms for real-time communication between and coordination of distributed BDI-based control components. State-of-the-art research on design of robotic control architectures presents interesting ideas for coordination and synchronization of distributed control components, which can guide the development of real-time distributed BDI architecture. The following briefly introduces two representatives of such approaches.

4.1 T-REX
T-REX [30] control architecture comprises a set of coordinated concurrent control loops named reactors, and a functional layer encapsulating a robot low-level functionalities. Reactors maintain their own view of the world and have their own control functionalities and temporal properties (i.e. lookahead window for deliberation and deliberation latency), therefore allow for partitioning a control problem in both functional and temporal horizons. T-REX has a central and explicit notion of time which allows execution of all reactors to be synchronized by an internal clock, ensuring the current state of the control system to be kept consistent and complete. The unit of time in T-REX is a tick, defined in external units on a per application basis. A deliberation time for each reactor in T-REX is bounded by its own deliberation latency, which is defined as a number of ticks. When a deliberation requires more than one tick, it should be defined as a proper sequence of steps to allow for interleaving synchronization (i.e. information exchange) in each tick.

4.2 ContrACT
Another example of distributed control architectures is ContrACT. The programming model of ContrACT [26] decomposes a robot control software into a set of controllable modules. Modules are independent real-time software tasks, which, depending on their types, use different communication models such as blocking/non-blocking and publish-subscribe/request-reply to communicate with each other. Some modules are reactive to events they receive, named asynchronous, and others are executed periodically, named synchronous. There is
also a single scheduler module, implementing a scheduling algorithm to schedule
the synchronous modules according to a set of constraints defined on the mod-
ule itself (e.g. duration of the execution) and on composition of modules (e.g.
precedence constraints, shared resources mutual exclusion, etc.). To achieve this
scheduling, the scheduler module works with operating system priorities and
activation requests sent to modules.

5 Sensory Data Processing

Today’s service robots are equipped with many sensors, providing them with dif-
ferent information about themselves and their environments. A robotic software
system is usually composed of different components such as face_recognition,
object_recognition and voice_recognition which process raw sensory data into
sensory information at different levels of abstraction to be used by the control
component(s) of a robot for decision making. To increase a robot’s level of in-
telligence and autonomy, the control component of the robot needs to aggregate
and correlate its multimodal sensory information to acquire and reason about
high-level information that describes the situation of the robot and its environ-
ment.

For a simple example of sensory management mechanisms necessary for a
control component, consider an extension of the task 5 of NAO robot in which
the user does not necessary need to introduce himself, but he can also just ap-
pear in front of NAO when giving the order. When NAO hears, “remember my
key is on the desk”, it considers the commanding user as the last one who has
introduced himself or has been appeared in from of its camera during the last
10 seconds. If nobody has introduced himself or appeared in front of the cam-
era in the last 10 seconds since an order has been given, the commanding user
is unknown and the command is ignored. The robot’s control component con-
tinuously receives events of recognized faces and phrases from face_recognition
and voice_recognition components. One can notice that providing such a sim-
ple functionality for the robot requires a sensory management mechanism to
allow for reasoning on different types of events and their temporal ordering and
properties.

Existing BDI-based agent programming languages provide limited support
for processing sensory inputs. These languages usually assume that information
is received from the environment in symbolic form, as required by the agent’s
deliberative components, and provide a simple event handling mechanism in
which a plan generating rule is applied to each event if applicable. They pro-
vide no tools or mechanisms for the management and processing of their input
data in order to acquire information at a higher level of abstraction. In order
to use the existing BDI-based agent programming languages to implement the
control mechanism of robotic systems, these languages should be extended with
a sensory component and its corresponding programming constructs to provide
different mechanisms necessary for the management and processing of the sen-
sory inputs. Sensory data processing and management is the focus of different
research. For example Buford et al. [6] propose an extension of BDI architecture with capabilities for event correlation and situation management and Heintz et al. [16] present various requirements of sensory data processing for bridging the sense-reasoning gap for autonomous robots. Based on these work, we can identify the following as some of important requirements for the sensory component.

– Garbage collection: the sensory component receives events from different event streams. Obviously it is not possible and also not efficient to keep all these events for infinite time. Events should be kept as far as necessary and automatically removed afterward. Determining the validity interval of events should be both possible manually by the programmer (e.g. by defining a time or length sliding window over a data stream) and also automatically by the system itself based on their usage in reasoning over high-level information.

– Knowledge representation: sensory component should provide a knowledge representation method for a unified representation of its different sensory inputs and necessary domain knowledge. It should be expressive enough to represent the knowledge of entities, their attributes including temporal information, and also structural and domain-specific relations between those entities.

– Reasoning: to process sensory data for recognizing high-level situations (i.e. events) appropriately for a robot’s tasks, a suitable reasoning mechanism should be provided that allows for manipulating micro-events based on the appearance, disappearance and reappearance of entities, constraints on their attributes and inter-relations, and the dynamic of entities (i.e. changes of entities’ attributes and interrelations). Also an ability for manipulation and reasoning about complex events and their temporal combinations are essential ingredients of such a reasoning mechanism.

– Event priorities: the order of processing events should be based on their priorities.

– Symbolic representation output: the acquired high-level information in the sensory component should be presented in the form of symbolic knowledge to be utilized by the deliberative component(s) of the robot control mechanism.

– Access methods: information in sensory module should be accessible by the control component both through querying (i.e. active perception) and receiving events (i.e. passive perception).

– Event handling: the event handling mechanism of agent programming languages should be enhanced to allow easy transfer of the knowledge from the sensory component into plans of a plan generating rules to be used for updating beliefs and goals.

6 Plan Execution Control

Current agent programming languages provide simple mechanisms for the execution control of a robot’s generated plans. Such mechanisms are often a combination of sequence, parallel, atomic, random order (AND), ordered choice (XOR),
random choice (OR), conditional choice (IF) and iteration (FOR, WHILE) plan operators. In order to facilitate the programming of autonomous robots, able to achieve complex goals in parallel, different mechanisms are necessary to deal with temporal and functional constraints related to a robot’s tasks and its physics, and for a proper parallel use of a robot’s resources.

Robotic research has developed many specialized execution languages to represent and execute plans that are generated manually by robotic software developers or automatically by planning systems [38]. Such languages provide many advanced mechanisms for synchronizing, coordinating and monitoring the executions of plans. This section discusses different plan execution control mechanisms needed by autonomous robots. A check list of such requirements is presented based on generalizing from the analysis of different plan execution control functionalities provided by TDL [34], PLEXIL [37], APEX [13], SMARTTCL [33], PRS [15] and PRS-lite [23] plan execution languages.

6.1 Representation of Complex Plans

To allow performing complex behaviors, different plan operators are needed for synchronizing the execution of actions/plans in complex arrangements, beyond the simple sequential and parallel settings provided by the existing agent programming languages. For example in the assistant robot application scenario, when NAO wants to check if there is enough drug in the box, it needs to go in front of the box (Location L), orient its head’s camera toward the box (Orientation O), and then take a picture to analyze if the box is empty or not. To achieve this goal in an efficient way, the NAO should be able to perform both actions of Move_To(L) and Orient_Head(O) in parallel, and then to take a picture only after both Move_To(L) and Orient_Head(O) actions have been successfully performed. Moreover it might be necessary for the camera to wait for a few second after the robot has arrived to the location and stopped walking, to start taking the picture. As can be seen from this example, developing cognitive robotic applications requires agent programming languages to be enriched with different mechanisms necessary for the synchronization of actions/plans executions in order and time. Current execution languages provide support for the following mechanisms:

- Hierarchical task decomposition: composing a complex plan from a set of other plans (i.e. subplans) in sequence and parallel orderings in different levels of a hierarchy.
- Controllability of the execution of a plan at different levels of its hierarchy.
- Supporting conditional contingencies, loops, temporal constraints and floating contingencies (i.e. event driven task execution) in the task tree decomposition: governing the execution of subplans (i.e. when to start, stop, suspend, resume/restart or abort a plan) by different conditions related to temporal constraints on the absolute time, constraints on execution status of other subplans, occurrence of certain events, constraints on a robot’s beliefs and also by direct access from other subplans (e.g. coordination using shared variables).
– Supporting both blocking and non-blocking intention dispatching for a new subgoal: the former places the new generated plan at the front of the execution path of the intention which generated the subgoal, the latter intends the new generated plan as a new intention.

– Control on expansion of a subplan such as complete expansion before execution or incremental expansion in runtime.

– Priority for execution of intentions.

– Supporting atomic and continuous actions (actions which provide feedback) in blocking and non-blocking modes.

### 6.2 Monitoring and Resource Management

A cognitive robot has different goals and receives different events. In a BDI architecture, the robot generates different plans to achieve those goals and react to those events. To provide a good level of autonomy and intelligence, it is obvious that a robot should be able to follow its different plans in parallel. For example when NAO is moving toward the drug box to check if it’s empty or not, it should be in the same time responsive to requests from its users (e.g. Task 5).

The problem is that execution of different plans in parallel can be conflicting due to a robot’s functional and resource constraints and should be coordinated based on the priorities of different plans. For example consider a use case in which NAO has picked up a piece of trash and going to put it into the trash can. Suddenly, NAO hears a user asking for help. To be able to help the user, NAO should go to the Red Button and have empty hands to press it. As can be noticed, this plan has two conflicts with the previous plan of the NAO (i.e. walking to the trash bin and having trash in hand). As helping the user is of the highest priority, NAO should leave the trash and start walking toward the Red Button immediately.

To facilitate the use of agent programming languages for implementing control systems of autonomous robots, these languages should be extended with different mechanisms and corresponding programming constructs necessary for coordinating the parallel execution of different plans. Moreover execution of plans should be monitored and their failures should be handled in a proper way. Current execution languages provide support for the following mechanisms:

– Monitoring the execution of a plan at different levels of its hierarchy.
– Monitoring different stages of the execution of a plan to guarantee its safe execution. Some conditions should be checked before starting/resuming the plan, some conditions should be checked continuously during the plan execution and some should be checked after finishing the execution of the plan.
– Monitoring different conditions such as temporal constraints on the absolute time, constraints on execution statuses of other subplans, occurrence of certain events and constraints on a robot’s beliefs.
– Representing and determining conflicts between different plans (e.g. explicit representation by denoting the resources they require or by providing shared variables and locking mechanisms).
Dealing with conflicts based on plans priorities and deadlines including dynamic prioritization and preemption.

Supporting different policies to deal with preempted and failed plans such as stopping, suspending or aborting a preempted plan or a failed plan.

Recovering from a plan failure and performing wind-down activities after suspension and before resuming a plan.

Furthermore, in addition to mechanisms needed for representing, reasoning on and resolving conflicts in plans execution, similar mechanisms are also necessary when processing goals and events to generate plans. For example, processing goals and events should be ordered based on their priorities and a lower priority goal conflicting with a higher priority goal might need to be considered as failed or to be kept to be processed later. These mechanisms are in particular needed in a distributed control paradigm in which a control component might receive conflicting goals from other components and should provide feedback to those components if it can handle those goals or not.

7 Conclusion

The paper presents various requirements for agent programming languages in order to implement cognitive robot control systems. These requirements are drawn partially based on an analysis of the problem at hand and partially based on a study of current autonomous robot control systems. We do not claim that the set of requirements presented in this paper is complete. However, by analyzing main aspects of programming cognitive robots and providing an overview of capabilities of current high-level robot programming tools, we have made a contribution to the systematic analysis and presentation of agent programming language requirements for programming cognitive robots. In this paper we limit ourselves to single robot applications. Multi-robot scenarios impose other requirements on agent programming languages for coordination, cooperation and communication between robots which have been left for further research.

The requirements presented in this paper show a big gap between capabilities of current agent programming languages and demands of programming autonomous robot control systems. Our research aim is to bridge this gap by extending the 2APL agent programming language to meet these requirements. One of such requirements is integration with existing robotic frameworks to facilitate controlling and communication with functional modules developed in these frameworks. This can encourage the use of agent programming languages by robotic community and facilitate their use for rapid prototyping and development of autonomous systems. To address this requirement, we have developed an environment interface for 2APL facilitating its integration with ROS using ROS communication mechanisms. We have used ROS to provide basic robotic capabilities such as face recognition, voice recognition and a number of high-level actions such as sit-down(), stand-up(), turn-neck(O) and walk-to(X,Y) for our NAO robots. Using 2APL and ROS, we have developed a demo application in
which different NAO’s movement can be controlled by voice. Also NAO can be
commanded to remember the face of a user and whenever a user greets NAO, if
NAO recognizes the user’s face, it greets the user by his/her name.

A part of the future work is to extend 2APL to provide support for develop-
ment of a sensory component for processing and management of heterogeneous
sensory information. This component should enable unified representation of
sensory data and domain knowledge and reasoning on high-level events (i.e. sit-
uations). The sensory information managed and processed by the sensory com-
ponent should be accessible by a BDI based control component in a symbolic
form and through both querying and receiving as events.

Another part of the future work is extending plan representation and ex-
ecution capabilities of 2APL. An extensive list of such required capabilities is
presented in section 6. These capabilities include providing support for govern-
ing the execution of plans by sequential, temporal and priority orderings, and
based on different internal conditions and external events, representing and han-
dling conflicts in parallel execution of plans, and monitoring and handling plans
execution failures.

Finally, we aim to provide support for development of distributed real-time
BDI-based control systems. This requires a specific version of 2APL dedicated to
development of real-time control components. The semantic and implementation
of such version should guarantee safe and bounded-time computations to enable
analysis and guaranteeing required real-time properties of a control component.
Also a dedicated architecture and runtime environment is required to support
the real-time coordination and communication of different control components of
a robot. We envision a ROS architecture consisting of a distributed set of control
component with different functionalities (e.g. deliberative, reactive, plan failure
handling) which can share beliefs and goals and other ROS components (i.e.
functional layer modules) including sensory components described above. These
components can have different real-time requirements. Some of them should be
run in real-time and guarantee bounded reaction and response time to events.
Our initial proposal for development of the real-time distributed architecture is
to use the Orocos Real-Time Toolkit which has been seamlessly integrated with
ROS.

References

language. In W. Vander Velde and J. W. Perram, editors, Agents Breaking Away
(LNAI), volume 1038 of LNAI, pages 4255. Springer-Verlag.
Report, Feb. 2000
3. Bordini, R. Braubach, L. Dastani, M. Fallah A.E. Gomez-Sanz, J. Leite, J, OHare,
forms for multi-agent systems. In Informatica 30, pages 3344
tems in AgentSpeak Using Jason. Wiley Series in Agent Technology
control architecture. 6th National Conference on Control Architectures of Robots
(2011) 16 p
uated Autonomous Systems. 9th Int. Conf. on Autonomous Agents and Multiagent
Systems (AAMAS), May 2010, Toronto, Canada.
In Proceedings of Second International Conference on Knowledge Representation
and Reasoning (KR91), pages 473484. Morgan Kaufmann.
ceedings of the First International Conference on Multi-Agent Systems (ICMAS-95),
pages 312-319, June 1995.
30. Rajan, K. Py, F. McGann, C. Ryan, J. OReilly, T. Maughan, T. and Roman, B.
(2009) Onboard Adaptive Control of AUVs using Automated Planning and Execu-
tion. Intl. Symposium on Unmanned Untethered Submersible Technology (UUST)
August 2009. Durham, NH
31. Ross, R.J. (2003) MARC - Applying Multi-Agent Systems to Service Robot Con-
trol, MSc Thesis, University College Dublin
on Open Source Software.
tional Reactive Task Execution in a Three Layer Architecture for Service Robots,
Int. Workshop on DYnamic languages for RObotic and Sensors systems (DY-
ROS/SIMPAR), Germany, pp. 274277.
Control. IROS.
35. Tsotsos, J. K. (1995) Behaviorist Intelligence and the Scaling Problem, Artificial
In Multi-agent programming: Languages, platforms and applications. Kluwer.
Execution Interchange Language (PLEXIL) for Executable Plans and Command
Sequences, Proceedings of the International Symposium on Artificial Intelligence,
Robotics and Automation in Space (i-SAIRAS).
38. Verma, V. Jonsson, A. Simmons, R. Estlin, T. Levinson, R. (2005) Survey of com-
mand execution systems for NASA spacecraft and robots. In Plan Execution: A
Reality Check Workshop at the International Conference on Automated Planning
and Scheduling (ICAPS)
An Agent-Based Cognitive Robot Architecture

Changyun Wei and Koen V. Hindriks
Interactive Intelligence, Delft University of Technology, The Netherlands
{C.Wei,K.V.Hindriks}@tudelft.nl

Abstract. We propose a new cognitive robot control architecture in which the cognitive layer can be programmed by means of the agent programming language GOAL. The architecture exploits the support that agent-oriented programming offers for creating cognitive robotic agents, including symbolic knowledge representation, deliberation via modular, high-level action selection, and support for multiple, declarative goals. The benefits of the architecture are that it provides a flexible approach to develop cognitive robots and support for a clean and clear separation of concerns about symbolic reasoning and sub-symbolic processing. We discuss the design of our architecture and discuss the issue of translating sub-symbolic information and behaviour control into symbolic representations needed at the cognitive layer. An interactive navigation task is presented as a proof of concept.

1 Introduction

The main motivation for our work is the need for a flexible, generic, high-level control framework that facilitates the development of re-taskable robot systems and provides a feasible alternative to the usual task- and domain-dependent development of high-level robot control. We believe that agent-oriented programming offers such an approach as it supports the programming of cognitive agents. Using agent programs to create the cognitive layer in a robot control architecture is natural and provides several benefits. It becomes relatively easy to adapt the control at the cognitive layer itself to various domains by exchanging one agent program with another. This approach is flexible and, if functionality of other layers is generic and can be used in multiple task domains, facilitates reuse. An agent-based approach, moreover, provides support for autonomous, reactive, and proactive behaviours and also endows a robot with the required deliberation mechanism to decide what to do next [1]. Of course, generality may come at a trade-off and does not imply that a generic architecture will always perform better than a dedicated robot control architecture [2].

Designing and developing a cognitive robot control architecture poses several challenges. Robots are embedded systems that operate in physical, dynamic environments and need to be capable of operating in real-time. A range of perception and motor control activities need to be integrated into the architecture. This poses a particular challenge for a cognitive, symbolic architecture as “it can be particularly difficult to generate meaningful symbols for the symbolic components of cognitive architectures to reason about from (potentially noisy) sensor
data or to perform some low-level tasks such as control of motors” [3]. Ideally, moreover, such an architecture should provide support for the integration or exchange of new and different sensors and behaviours when needed. Given the complexity and the number of components needed in a robot control architecture, one also needs to consider how all the processing components in the system communicate and interact with each other [4].

In this paper, we propose an agent-based cognitive robot control architecture that combines low-level sub-symbolic control with high-level symbolic control into the robot control framework. We use the agent programming language GOAL [5, 6] to realize the cognitive layer, whereas low-level execution control and processing of sensor data are delegated to components in other layers in the proposed architecture. GOAL, among others, supports the goal-oriented behavior and the decomposition of complex behavior by means of modules that can focus their attention on relevant sub-goals. GOAL has already been successfully applied to control real-time, dynamic environments [7], and here we demonstrate that it also provides a feasible approach for controlling robots. In our approach, the cognitive layer is cleanly separated from the other layers by using the Environment Interface Standard (EIS; [8]).

The paper is structured as follows. Section 2 briefly discusses some related work. Section 3 presents and discusses the design of the cognitive robot architecture. Section 4 presents a proof of concept implementation. Section 5 concludes the paper and discusses future work.

2 Related Work

Cognitive robots are autonomous and intelligent robot systems that can perform tasks in real world environments without any external control, and are able to make decisions and select actions in dynamic environments [9]. Here we discuss some related work that either explicitly aims to develop a cognitive architecture or uses some kind of symbolic representation for controlling a robot.

The work that is most similar in spirit to our own is that of [10] and [11]. In [10] the high-level language Golog is used for controlling a robot. Golog supports writing control programs in a high-level, logical language, and provides an interpreter that, given a logical axiomatization of a domain, will determine a plan. Golog, however, does not provide a BDI perspective on programming agents, and in [10] it does not discuss the robot control architecture itself. A teleo-reactive program, consisting of multiple prioritized condition-action rules, is used to control a robot in [11] the proposed robot architecture in is not discussed in any detail.

CRAM [1] is a software toolbox designed for controlling the Rosie robot platform developed at the Technische Universität München. It makes use of Prolog and includes a plan language that provides a construct for concurrent actions. The CRAM approach also aims at providing a flexible alternative to pre-programmed robot control programs. The main difference with our approach is that the reasoning and planning components are separated.
It has been argued that building robot systems for environments in which robots need to co-exist and cooperate with humans requires taking a *cognitive stance* [12]. According to [12], translating the key issues that such robot systems have to deal with requires a cognitive robot control architecture. Taking a cognitive stance towards the design and implementation of a robot system means that such a system needs to be designed to perform a range of *cognitive functions*. Various *cognitive architectures*, such as ACT-R [13] and SOAR [14], have been used to control robots. These architectures were not primarily aimed at addressing the robot control problem and in this sense are similar to agent programming languages, the technology that we advocate here for controlling robots. SOAR has been used to control the hexapod HexCrawler and a wheeled robot called the SuperDroid [3]. ADAPT (Adaptive Dynamics and Active Perception for Thought) is a cognitive architecture based on SOAR that is specifically designed for robotics [15]. The SS-RICS (Symbolic and Sub-symbolic Robotic Intelligent Control System) architecture for controlling robots is based on ACT-R; SS-RICS is intended to be a theory of robotic cognition based on human cognition [16, 2]. Unlike [17], we are not mainly concerned here with the long-term goal of developing robotic systems that have the full range of cognitive abilities of humans based on a cognitive architecture such as SOAR. Our work is oriented towards providing a more pragmatic solution for the robot control problem as discussed above. Although our work may contribute to the larger goal that [17] sets (as there are quite a few similarities between BDI-based agents and cognitive architectures such as ACT-R and SOAR), we do not address this question.

3 Cognitive Robot Control Architecture

This section introduces our cognitive robot control architecture. We discuss several issues including the processing and mapping of sensory data to a symbolic representation, the translation of high-level decisions into low-level motor control commands, and the interaction of the various architecture components.

3.1 Overall Design of the Architecture

Figure 1 shows the main components of our architecture, including a symbolic, cognitive layer (GOAL), a middle layer for controlling robot behavior (written in C++), and a hardware control layer (using URBI\(^1\), a robotic programming language). The Environment Interface Standard (EIS) layer provides the technology we have used to manage the interaction between the symbolic and sub-symbolic layers. EIS provides a tool to deal with the issue that *sub-symbolic* sensory data is typically noisy, incomplete, quantitative measurements, whereas the symbolic layers need the *symbolic representation* that supports logical reasoning.

The main functions for controlling a robot are placed in the behavioural control layer. In addition to the functions such as (object) recognition, navigation,

\(^1\) http://www.urbiforge.org/
localization, path planning and other common functions, this layer is also responsible for communicating with the higher-level symbolic components, including the interpretation of symbolic messages that represent actions and making the robot perform these actions in its physical environment.

The EIS layer acts as a bridge between the behavioural and cognitive control layers. Because these layers use different languages for representing sub-symbolic and symbolic information, respectively, we need an interface to translate between these representations. The cognitive control layer acts as a task manager for the robot and provides support for managing the robot’s mental state which allows the robot to keep track of what is happening while executing a task.

3.2 System Architecture and Components

A robot control architecture provides an organizational structure for software components that control a robotic system [18]; such architectures are specialized because of the unique requirements that embedded systems such as robots impose on software. Here we discuss the detailed architecture presented in Figure 2 and also discuss some of the implementation details.

Robot Platform The architecture has been implemented on the humanoid Nao robot platform from Aldebaran Robotics. We use the URBI [19] middleware that provides the urbiscript language for interfacing with the robot’s hardware. We have chosen URBI instead of the Robot Operating System (ROS) platform

\[\text{http://www.aldebaran-robotics.com/}\]
\[\text{http://www.ros.org/}\]
because it does not include the orchestration layers present in URBI that provide support for parallel, tag-based, and event-driven programming of behaviour scripts. When the architecture is executed, an urbiscript program, which initializes all API parameters of sensors and motors (e.g., the resolution and frame rate of camera images) is executed to configure the robot platform.

**Behavioural Control** The behavioural control layer is written in C++, connecting the robot hardware layer with higher layers via a TCP/IP connection. This layer is mainly responsible for information processing, knowledge processing, communication with the deliberative reasoning layer and external robots, and action and behaviour executions. All of these components can operate concurrently. The main functional modules involve:

- **Sensing**, for processing sensory data and receiving messages from other robots. Sensors that have been included are sonar, camera, microphone, an inertial sensor, and all sensors monitoring motors. Because the memory space required for camera images is significantly bigger than that for other sensors, the transmission of images is realized via a separate communication channel.
– **Memory**, for maintaining the *global memory* and *working memory* for the behavioural layer. In global memory, a map of the environment and properties of objects or features extracted from sensor data are stored. We have used a 2D grid map for representing the environment. This map also keeps track which of the grid cells are occupied and which are available for path planning. The working memory stores temporary sensor data (e.g., images, sonar values) that are used for updating the global memory.

– **Information Processor**, for image processing. This component provides support for object recognition, feature extraction and matching, as well as for information fusion. Information fusion is used to generate more reliable information from the data received from different sensors. We have used the OpenCV [20] library to implement algorithms and methods for processing images captured by the camera. Algorithms such as Harris-SIFT [21], RANSAC [22], and SVM [23] for object recognition and scene classification have been integrated in this module.

– **Environment Configurator**, for interpreting and classifying events that occur in the environment. For example, when the output of the left sonar exceeds a certain value, this module may send a corresponding event message that has been pre-configured. This is useful in case such readings have special meaning in a domain. In such cases, an event message may be used to indicate what is happening and which objects and features such as human faces and pictures have been detected.

– **Navigation** includes support for *Localization*, *Odometry*, *Path Planning* and *Mapping* components, which aid robots in locating themselves and in planning an optimal path from the robot’s current position to destination places in a dynamic, real-time environment. Once a robot receives a navigation message from the cognitive layer with an instruction to walk towards the destination in the 2D grid-map space, the *Path Planning* component calculates an optimal path. After each walking step, *Odometry* will try to keep track of the robot’s current position, providing the real-time coordinates of the robot for planning the next walking step. Due to the joint backlash and foot slippage, it is impossible to accurately estimate the robot’s position just by means of odometry. For this reason, the robot has been equipped with the capability to actively re-localize or correct its position by means of predefined landmarks. Additionally, navigation requires back-and-forth transformation between coordinate systems. The odometry sensors, for example, provide the robot’s coordinates in a World Coordinate System (WCS) that needs to be mapped to the robot’s position in 2D Grid-map Coordinate System (GCS) for path planning. However, when executing walking steps, the position in GCS has to be transformed into the Local Coordinate System (LCS) that the robot uses.

– **Communication**, for delivering perception messages to the EIS component and receiving action messages from the EIS. The communication component is constructed based on a Server/Client structure and uses the TCP/IP protocol. The behaviour layer in a robot starts a unique server to which
a cognitive layer can connect as a client (thus facilitating the swapping of control from one robot to another).

- **Debugger Monitor** provides several GUIs that are useful for debugging robot programs, enabling developers to visualize sensory data and allowing them to set specific function parameters. This component also includes a *Wizard of Oz interface* to conduct human-robot interaction experiments.

- **Action Execution**, for instructing a robot to perform concrete behaviours and actions. The actions include motion movements such as walking and turning around while the behaviours include predefined body gestures such as sitting down, standing up and raising the arms of humanoid robots.

*Environment Interface*  The environment interface layer built using EIS [8] provides support for interfacing the behaviour layer with the cognitive layer. The core components in this layer include an *Environment Model* that establishes the connection between cognitive robotic agents with external environments, an *Environment Management* component that initializes and manages the interface, and an *Environment Interface* component that provides the actual bridge between a cognitive agent and the behavioural layer.

*Deliberative Reasoning and Decision Making*  The cognitive control layer provides support for reasoning and decision making. In our architecture, we employ the GOAL agent programming language for programming the high-level cognitive agent that controls the robot. GOAL is a language for programming logic-based, cognitive agents that use symbolic representations for their beliefs and goals from which they derive their choice of action. Due to space limitations, we do not describe GOAL agent programs in any detail here but refer the interested reader for more information to [5, 6].

3.3 Information & Control Flow in the Architecture

A key issue in any robot control architecture concerns the flow of information and control. Each component in such an architecture needs to have access to the relevant information in order to function effectively. In line with the general architecture of Figure 1 and layered architectures in general, different types of data are associated with each of the different layers, where EIS is a mediating layer between the behavioural and cognitive layer. At the lowest level all “raw” sensor data is processed yielding information that is used in the behavioural layer. The information present in the behavioural layer is abstracted and translated into knowledge by means of EIS that is used in the cognitive layer. Actions in turn are decided upon in the cognitive layer. These actions are translated by EIS into behaviours that can be executed at the behavioural layer. Finally, these behaviours are translated into motor control commands at the lowest layer.

*Bottom-up Data Processing and Top-down Action Execution*  The processing of data starts at the bottom in the architecture and follows a strict bottom-up processing scheme whereas for action execution the order is reversed and a strict
top-down scheme is followed. At each of the layers different types of representa-
tions are used: urbiscript at the lowest, C++ data structures in the behavioural
layer, and Prolog representations are used in the logic-based cognitive layer.
Data processing starts by capturing values from various sensors. This sensory
data then is analysed to extract useful features and to classify the data using
basic knowledge of the environment. Fusion of different data sources is used to
increase reliability. Finally, by using more high-level, semantic knowledge about
the environment, symbolic representations are obtained by combining, abstract-
ing and translating the information present at the behavioural layer by means
of the EIS layer. The EIS layer provides symbolic “percepts” as input to the
cognitive layer. Different aspects of the environment are handled by different
 techniques. For example, in order to recognize human faces, camera images need
to be used and processed. First, several image frames need to be captured by
the lowest layer for further processing in the behavioural layer which runs face
detection or recognition algorithms. Using information obtained about the ex-
act regions where a face is located in an image, further processing is needed to
match these regions with a database of faces. Upon recognition, finally, a face
can be associated with a qualitative description, i.e. a name, which is sent to the
cognitive layer. The cognitive layer then uses this information and decides e.g.
to say “Hello, name.”

The cognitive layer is the center of control and decides on the actions that
the robot will perform. Actions are translated into one or more possibly complex
behaviours at the behavioural layer. Of course, such behaviours, and thus the
action that triggered these, take time and it is important to monitor the progress
of the behaviours that are used to execute a high-level action. The cognitive
layer delegates most of the detailed control to the behavioural layer but needs
to remain informed to be able to interrupt or correct things that go wrong
at this layer. One clear example concerns navigation. At the cognitive layer a
decision to perform a high-level action goto(roomA) may be made which then
is passed on to a navigation module at the behavioural layer. The latter layer
figures out details such as the exact destination position of roomA on the 2D grid
map and then plans an optimal path to the destination. Due to the unreliability
of walking (e.g. foot slippage), the optimal path needs to be continuously re-
evaluated and re-planned in real-time. This happens in the behavioural layer.
However, at the same time, the cognitive layer needs to monitor whether the
planning at the behavioural layer yields expected progress and may interrupt by
changing target locations.

Synchronization of Perception and Action Part of the job of the EIS intermediate
layer is to manage the “cognitive” load that needs to be processed in the cognitive
layer. One issue is that the Environment Configurator in the behavioural layer
typically will produce a large number of (more or less) identical percepts. For
example, assuming an camera image frame rate of 30 fps, if a robot stares at an
object for only one second, this component will generate 30 identical percepts.
In order to prevent that a stream of redundant percepts is generated the EIS
layer acts as a filter. However, this only works if in the behavioural layer objects
can reliably be differentiated and some assumptions about the environment can be made.

Another issue that needs to be handled by the EIS layer concerns the actions that are sent by the cognitive layer to the behavioural layer. Because behaviours have duration and the cognitive layer runs in parallel with the behavioural layer, the cognitive layer may send the same action multiple times to the behavioural layer or may send another action while an earlier action has not yet been completed. Because only some actions can be run in parallel, a decision may have to be made about which action has priority. For example, a robot can walk and talk at the same time but this is not the case for walking and sitting down. In the latter case, the behavioural layer needs to be able to gracefully terminate one behaviour in favour of another, or, in exceptional cases, ignore an action command from the cognitive layer.

3.4 Other Distinctive Properties of Our Architecture

In this section, we briefly discuss some distinctive properties of our architecture.

Decoupled Reactive and Deliberative Control Similar to most hierarchical architectures, we also distinguish reactive and deliberative control in our architecture. As usual, reactive control resides in the behavioural layer that is responsible for sub-symbolic information processing and behaviour execution, whereas deliberative control resides in the cognitive layer that is responsible for symbolic reasoning and decision making. In fact, these layers have been rigidly separated in our architecture by means of the EIS layer. In other words, in our design these layers have been completely decoupled. The main benefit of such a design consist of a clean separation of concerns. The price that needs to be paid for this strict separation is duplication of information and an additional effort of developers to design a well-defined interface between the two layers.

Decoupling of these layers facilitates the more or less independent programming of high-level, cognitive agents that control a robot as well as of lower-level behavioural functions. An agent programmer, for example, does not need to spend time handling object recognition. Similarly, a behaviour programmer who codes in C++ does not have to consider decision making nor even master the agent programming language used in the cognitive layer.

Multi-Goal Maintenance Often when robots try to accomplish a complex task, they will have to fulfil multiple goals [28]. Robots, however, cannot perform all of these goals at the same time either because goals may conflict with each other, or achieving a goal presupposes the achievement of others. As a simple example, when a robot needs to execute a command "Pick up object B in place A", it actually has to fulfil two sub-goals in order to accomplish the goal; the robot needs to figure out that it first has to perform the action goto(place(A)) and thereafter perform pickup(object(B)).
The cognitive layer realized by an agent programming language such as GOAL is able to handle multiple, possibly conflicting goals. GOAL [5] has a special operator to express that a goal needs to be achieved: \(a\text{-}\text{goal}(\varphi)\) as well as that (part of) a goal has been achieved \(\text{goal}\text{-}\text{a}(\varphi)\). For instance, \(\text{goto}(\text{place}(A))\) can be expressed as an achievement goal as follows:

\[
a\text{-}\text{goal}(\text{goto}(\text{place}(A))) \overset{df}{=} \text{goal}(\text{goto}(\text{place}(A))), \text{not} (\text{bel}(\text{in}(\text{place}(A)))
\]

Goal achievement can be expressed by:

\[
\text{goal}\text{-}\text{a}(\text{goto}(\text{place}(A))) \overset{df}{=} \text{goal}(\text{goto}(\text{place}(A)), \text{bel}(\text{in}(\text{place}(A)))
\]

The handling of multiple goals has been demonstrated already successfully in real-time gaming environments [7].

**Multi-Modal Communication** Our architecture endows robots with multiple channels of communication with humans or other robots.

- **Semantic Representation for Human Robot Interaction** Since one of the objectives of developing intelligent robots is to offer better lives to humans, robots should be able to operate and help humans in daily activities. When interacting with humans, robots are required to incorporate communication and collaboration abilities. Human-robot interaction imposes special requirements on robot systems, and such interaction should be natural and intuitive for a human user. In particular, human-oriented interaction needs to take human abilities for effectively interacting into account. Symbolic knowledge representations could be accessible for human interpretation through precept-symbol identification by the designer [29]. Generally speaking, robot control systems need to adapt to humans by facilitating interaction that humans can understand. Our architecture supports knowledge processing and can produce semantic messages to facilitate interaction at the knowledge level with humans.

- **Shared Perceptions and Beliefs for Multi-Robot Interaction** From the perspective of multi-robot systems, robot systems are seldom stand-alone systems; however, robots should share their perceptions in order to interact with each other. For an individual robot, the perception messages will be transmitted from the sub-symbolic layers to the symbolic layers via an inner-communication channel, where the connection is constructed based on Server/Client structure in the TCP/IP protocol. The sub-symbolic layer in each robot starts a server that is unique for that robot, whereas the symbolic layer itself can have several clients, each of which can connect to either the server of its own or other robots. In this way, the robots cannot only receive the information from its own perceptions but also exchange their perceptions with other robots.

Our architecture provides the communication mechanisms for both inner-communication and external-communication, enabling robots to send and receive perception messages, and exchange mental beliefs present in the cognitive layer.
4 Experiment

To evaluate the feasibility of the proposed architecture, we have carried out a simple navigation task using a humanoid robot Nao (See Figure 3) built by Aldebaran Robotics. In this task, the Nao robot acts as a message deliverer which is supposed to enter the destination room and deliver a message to the people in the room.

![Fig. 3. Humanoid robot Nao](image1.png) ![Fig. 4. The main GUI of the robotic behavior layer](image2.png)

The experiment is carried out in a domestic corridor environment. A predefined map has been built for the robot to localize itself. The robot begins walking from the starting place, and the goal of the robot is to enter the destination room, to show an arm-raising gesture, and eventually to deliver a coffee-break message. The behavioural layer processes the lower level functions; for example, Localization is used to estimate the robot’s position, and real-time path planning is used to steer the walking of the robot. Meanwhile, it will communicate with and provide perceptual knowledge to the high-level cognitive layer. Figure 4 shows the main GUI of the behavioural layer, and Figure 5 shows the GUI for the cognitive layer implemented in Goal.

The navigation task begins with a message navigation broadcasted from the behavioural layer to the cognitive layer, which indicates that both the low-level control and high-level control are ready to start. Then, the first action command goto(roomA) is generated. The Navigation module in the behavioural layer takes charge and steers the walking from the starting place to the destination room. When the robot arrives at the door of the destination room, a percept at(roomA) is sent to the cognitive layer that will generate the next action enterRoom(roomA). After the robot enters the destination room, another percept in(roomA) is sent. Thereafter, when the robot has the belief of being in the destination room, it will perform the actions: showing gesture(raiseArm)
and saying "Hi, let's take a coffee break!". In Figure 5, the action logs have been printed out to illustrate the decision making related to the task.

Although this navigation task is simple, it shows that uncertain, quantitative sensory data is mapped into symbolic perceptual knowledge and allows the robot to select their actions while performing tasks in a real world environment.

5 Conclusion and Future Work

The navigation task that the robot has performed in the above section demonstrates the feasibility of using a cognitive layer to control physical robots by means of agent-oriented programming. It also demonstrates that the clean separation of sub-symbolic and symbolic layers via the EIS component is feasible. Although our general architecture is similar to the layered approaches that have been used in many robot projects [24–27], we believe that the use of EIS provides a more principled approach to manage the interaction between sub-symbolic and symbolic processors. Of course, it is important that the cognitive layer (agent program) needs adequate perceptions to make rational decisions given its specific environment. Our architecture provides sufficient support from various components dealing with Perception, Knowledge Processing, and Communication to ensure this.

Future work will concentrate on applying our proposed architecture for multirobot teamwork in the Block World for Teams environment [30], in which each robot can exchange their perceptions and share their mental states in the cognitive layer to coordinate their actions. To this end, in the low-level layers, we will integrate more image processing functions to endow robots with more perception inputs. Besides, we will also develop the adaptive ability so that robots can
learn or gain knowledge from experience. We believe this learning mechanism will facilitate robots to adjust to or understand environments more efficiently and reliably in the future.

References

A Programming Framework for Multi-Agent Coordination of Robotic Ecologies

M. Dragone¹, S. Abdel-Naby¹, D. Swords¹, and M. Broxvall²

¹ University College Dublin, Dublin, Ireland,
mauro.dragone@ucd.ie,
² Örebro University, Fakultetsgatan 1, SE-70182, Örebro, Sweden

Abstract. Building smart environments with robotic ecologies made up of distributed sensors, actuators and mobile robot devices extends the type of applications that can be considered, and reduces the complexity and cost of such solutions. While the potentials of such an approach makes robotic ecologies increasingly popular, many fundamental research questions remain open. One such question is how to make a robotic ecology self-adaptive, so as to adapt to changing conditions and evolving requirements, and consequently reduce the amount of preparation and pre-programming required for their deployment in real world applications. In this paper we present a framework for integrating an agent programming system with the traditional robotic and middleware approach to the development of robotic ecologies. We illustrate how these approaches can complement each other and how they provide an avenue where to pursue adaptive robotic ecologies.

Keywords: robotic ecologies, multiagent systems, agent and component based software engineering

1 Introduction

This paper describes the integration between an agent programming system and a middleware supporting the development of Robotic Ecologies - networks of heterogeneous robotic devices pervasively embedded in everyday environments.

Robotic ecologies is an emerging paradigm, which crosses the borders between the fields of robotics, sensor networks, and ambient intelligence (AmI). Central to the robotic ecology concept is that complex tasks are not performed by a single, very capable robot (e.g., a humanoid robot butler), instead they are performed through the collaboration and cooperation of many networked robotic devices (including mobile robots, static sensors or actuators, and automated home appliances) performing several steps in a coordinated and goal oriented fashion.

One of the key strengths of such an approach is the possibility of using alternative means to accomplish application goals when multiple courses of actions are available. For instance, a robot seeking to reach the user in another room may decide to localize itself with its on-board sensors, or to avail itself of the more accurate location information from a ceiling localization system.
However, while having multiple options is a potential source of robustness and adaptability, the combinatorial growth of possible execution traces makes it difficult to scale to complex ecologies. Adapting, within tractable time frames, to dynamically changing goals and environmental conditions is made more challenging when these conditions fall outside those envisioned by the system designer.

In the EU FP7 project RUBICON (Robotic UBIquitous COgnitive Network) [1][2] we tackle these challenges by seeking to develop goal-oriented robotic ecologies that exhibit a tightly coupled, self-sustaining learning interaction among all of their participants. Specifically, we investigate how all the participants in the RUBICON ecology can cooperate in using their past experience to improve their performance by autonomously and proactively adjusting their behaviour and perception capabilities in response to a changing environment and user needs.

An important pre-requisite of such an endeavour, which is addressed in this paper, is the necessary software infrastructure subtending the specification, integration, and the distributed management of the operations of robotic ecologies. Specifically, this work builds upon the Self-OSGi [3][4], a modular and lightweight agent system built over Java technology from the Open Service Gateway Initiative (OSGi) [5], and extends it to operate across distributed platforms by integrating it with the PEIS middleware, previously developed as part of the Ecologies of Physically Embedded Intelligent Systems project [6]. The result described in this paper is a distributed programming framework for the specification and the development of robotic ecologies.

The remainder of the paper is organized in the following manner: Section 2 provides an overview of the state of the art techniques for the coordination of robotic ecologies, with an emphasis on those pursued within the PEIS initiative - the starting point for the control of RUBICON robotic ecologies. Section 3 presents the Self-OSGi component & service-based agent framework, and the way it has been recently extended and integrated with PEIS. Section 4 illustrates the use of the resulting multi-agent framework with a robotic ecology experiment. Finally, Section 5 summarizes the contributions of this paper and points to some of the directions to be explored in future research.

2 PEIS

The PEIS kernel [7] and related middleware tools are a suite of software, previously developed as part of the PEIS project [6] in order to enable communication and collaboration between heterogeneous robotic devices.

The PEIS kernel is written in pure C (with binding for Java and other languages) and with as few library and RAM/processing dependencies as possible in order to fit on a wide range of devices.

PEIS includes a decentralized mechanism for collaboration between separate processes running on separate devices that allows for automatic discovery, high-level communication and collaboration through subscription based connections. It also offers a shared, tuple space blackboard that allows for high level collaboration and dynamic self-configuration between different devices through the
exchange and storage of tuples (key-value pairs) used to associated any piece of data (and related meta-information, such as timestamps and MIME types), to a logical key.

### 2.1 Tuples and Meta-Tuples

A tuple’s key in PEIS consists of two parts: *(name, owner)* where *name* is a string key for the tuple, and *owner* is the address (id) of a PEIS responsible for the tuple.

In the most simple scenario for executing a collaboration between components, producers create data in their own tuple space and consumers establish subscriptions to these tuples to access the data to be used.

However, consuming components cannot know in advance from where to read the data to be used. Meta tuples are a mechanism addressing such a problem in a general way. By using these as inputs it is possible for consumers to read hard coded meta tuples from their own tuplespace. This corresponds to meta tuples acting as named input ports in other middleware.

To configure such a consumer, a configuration writes the id and key of tuples produced by any producer. The consumer will then automatically subscribe to and read the data from the producer, as in the following (pseudo-code) example.

**Producer 42:**
```
while (true):
    setTuple "temperature" <= sensorReading()
```

**Consumer 22:**
```
subscribeIndirectTuple(peisId(), "heat")
while true:
    T = findIndirectTuple("heat")
    ... do something with T ...
```

**Configurator:**
```
setTuple "22.heat" <= "META 42 temperature"
```

### 2.2 PEIS-init

PEIS relies on a program called PEIS-Init to act as central location on each host to start or stop and monitor the execution of functional components. To this end, PEIS-init component relies on a set of component files on the local machine to determine which component can run on it and what their semantic descriptions are. For each possible component, PEIS-init subscribes to tuples to set the start, stop or restart state of the components. It forks and executes the corresponding software components if the components are requested to be run (i.e. when their *reqState* tuples are set to value *on*), monitors their inputs, outputs and execution states (restarting them if necessary) and stops the components when they are no longer needed (when their *reqState* tuple is set to *off*).

### 2.3 Action & Configuration Planning for Robotic Ecologies

The general problem of self-configuration of a distributed system is addressed in several fields, including ambient intelligence [8][9], web service composition...
These works, however, do not address the same type of problem considered here: functional coordination of a robotic ecology, in which the components of the system exchange continuous streams of data and can interact with the physical world.

Although classical AI planning such as STRIPS based operators can easily be extended to take the role of high-level coordinators for robotic ecologies [13] [14], these planning methods suffer from a number of challenges that make them less than ideal.

The first of these challenges is due to the demands on robustness combined with a demand of combinatorial generality where the possible combinations of devices should be able to provide functionalities to assist other devices is expected to grow superlinearly as more devices are added to the ecology. Computing which actions are to be performed by individual devices are traditionally delegated to an action planner that reasons about the possible outcomes of different actions on a given model of the environment. As the environments become increasingly complex, unstructured and with increasing demands of methods for accurately handling errors in perception or actuation these planning models tend to increase in complexity and to become intractable.

We call the set of devices that are actively exchanging data in a collaborative fashion at any given time the configuration of the ecology. The task of computing the configuration to be used at any given time in order to accomplish the actions generated by an action planner can also be modelled explicitly as a search problem and solved either in a dedicated configuration planner or as an integral step of the action planners. For this purpose such configuration planners typically rely on introspection and semantic descriptions of the available components in order to create a domain description that includes all the available devices and the actions and data-exchange functionalities that they support. This is illustrated Figure 2 where a configurator plans for a subset of the available devices to perform specific localization tasks in order to assist the robot Astrid to navigate and open a refrigerator door.

Fig. 1. A simple PEIS-Ecology (taken from [6]). Left: The ceiling cameras provide global positioning to the robot. The robot performs the door opening action by asking the refrigerator to do it. Right: Corresponding functional configuration of the devices involved
Although some success \cite{13} \cite{14} have been made in the literature for solving the joint action and configuration planning problems at a global level for heterogeneous robotic ecologies or to enable their purely reactive configuration \cite{15}, we propose to distribute certain aspects of these tasks over a number of agents with varying responsibilities and functionalities in order to lead to better scalability, robustness and fault tolerance.

Agent and Multi Agent Systems (MASs) are regarded as a general-purpose paradigm to facilitate the co-ordination of complex systems built in terms of loosely-coupled, situated, autonomous and social components (the agents). In particular, the Belief Desire Intention (BDI) agent model, used in the Self-OSGi framework discussed in the following section, provides a simple but extensible model of agency that explicitly addresses the production of rational and autonomous behaviour by agents with limited computational resources.

3 Self-OSGi

Self-OSGi \cite{3} \cite{4} is a BDI agent framework built on OSGi Java technology and purposefully designed to support the type of collaboration envisaged within the ubiquitous and embedded systems targeted by RUBICON.

The component & service orientation used in the design of Self-OSGi is an highly popular, mainstream approach used to build modular software systems. Component & service frameworks operate by posing clear boundaries (in terms of provided and required service interfaces) between software components and by guiding the developers in re-using and assembling these components into applications. Self-OSGi addresses the lack of common adaptation mechanisms in these frameworks by leveraging their previously unexploited similarities with the BDI agent model.

3.1 OSGi

OSGi specification \cite{5} is currently the most widely adopted technology for building modular control systems for networked home applications, with many implementations targeting computationally constrained platforms. Within the AAL domain, OSGi-based middleware have long been used to provide the technical basis for integrating network devices and services, e.g. in EU projects such as Amigo, OASIS, SOPRANO, and their recent consolidation in the UniversAAL platform.

OSGi defines a standardised component model and a lightweight container framework, built above the JVM, which is used as a shared platform for network-provisioned services and components specified through Java interfaces and Java classes. Each OSGi platform facilitates the dynamic installation and management of units of deployment, called bundles, by acting as a host environment whereby various applications can be executed and managed in a secured and modularised environment. An OSGi bundle is packaged in a Jar file and organises the frameworks internal state and manages its core functionalities. These
include both container and life cycle operations to install, start, stop and remove components as well as checking dependencies.

The separation between services and their actual implementations is the key to enable system adaptation. With OSGi, in addition to syntactic matching of service interfaces, developers can also associate lists of name/value attributes to describe the semantic of each service, and use the LDAP filter syntax to search the services that match given search criteria. Furthermore, Declarative Services (DS) for OSGi offers a declarative model for managing multiple components within each bundle and also for automatically publishing, finding and binding their required/provided services, based on XML component definitions. However, DS only matches pre-defined filters with pre-defined services attributes of already active components, but does not consider the automatic instantiation of new components, the context-sensitive selection of their services, or the automatic recovery from their failure - all necessary features for the construction of context-aware, adaptive systems.

3.2 Component & Service based Agent Model

Self-OSGi addresses the issues outlined in the previous section by translating the BDI agent model [16] into general component & service concepts. In particular, the separation between the services interface and the services implementation is the basis for implementing both the declarative and the procedural components of BDI-like agents, and also for handling dynamic environments, by replicating their ability to search for alternative applicable plans when a goal is first posted or when a previously attempted plan has failed.

Belief Model: As in the Jadex agent language [17], Self-OSGi represents beliefs as simple Java objects. Compared to agent toolkits where beliefs are stored as logic predicates, objects have the advantage of providing a strong typed definition of agent’s beliefs. In addition, within Self-OSGi, a belief set is implemented as a Belief Set component with clearly defined interfaces, which are used to access any data that may affect the value of its beliefs.

Service Goal Model: Goals, describing the desires that the agent may possibly intend, are represented in Self-OSGi by the (Java) interfaces of the services that may be used to achieve them, or service goals.

Service goals may represent either: (i) performative sub-goals defining the desired conditions to bring about in the world and/or in the systems state - for instance, the method "(void) beAt(X, Y)" in the goal service GoalNavigation may be used by a robotic agent to represent the goal of being at a given location - and (ii) knowledge sub-goals subtending the exchange of information. For instance, the method "Image getImage()" in the GoalImage service goal may be used to express the goal of retrieving the last video frame captured by one camera. In addition, service goals attributes may be used to further characterise each service goal, e.g. the characteristic of the information requested/granted,
as well as important non-functional parameters. In particular, attributes may be used to identify the entity (a specific robot agent or part of it) responsible of some perceptual and/or acting process. For instance, the attribute Agent may be used to represent the name of the robot providing video frames, while the attribute Side may be used to specify to which one of the robotic cameras (left or right eye) the video corresponds to.

**Component Plan Model:** A plan, describing the means to achieve a goal (its post-condition), is represented by the component - component plan - implementing (providing) it. A component plan may require a number of service goals in order to post sub-goals, to perform actions, and also to acquire the information it needs to achieve its post-condition. For instance, a Navigator component plan may process the images from a robot's camera and control the velocity and the direction of the robot to drive it safely toward a given location. The same component plan may subscribe to range data from a laser sensor, to account for the presence of obstacles on its path.

**Semantic Descriptions:** Self-OSGi re-uses the OSGi XML component descriptions and enriches them with properties guiding its agent-like management of components’ dependencies and components’ instantiation.

As a way of example, the following is part of the XML documents describing a Navigation, a CameraLocalization and a LaserLocalization component plans.

In order to clarify its correspondence with the BDI model it represents, each XML is preceded with a comment in the form $e: \Psi \leftarrow P$ where $P$ is the body of the plan, $e$ is the event that triggers the plan (the plan’s post-conditions), and $\Psi$ is the context for which the plan can be applied (which corresponds to the preconditions of the plan).

```
GoalNavigation(?Agent) : true ← {achieve(GoalLocalization(?Agent)); ...

<scr:component ... factory="Navigation" name="Navigator">
<implementation class= "Navigator"/>
<property name="?Agent" type="String" value="The name of the robot supposed to move">
<service>
<provide interface="GoalNavigation"/>
</service>
</scr:component>

GoalLocalization(?Agent) : (light > 30) ← {achieve(GoalVideo(?Agent)); ...

<scr:component ... factory="CameraLocalization" name="CameraLocalization">
<implementation class= "CameraLocalization"/>
<property name="?Agent" type="String" value="The name of the robot to be localized">
<service>
<provide interface="GoalLocalization"/>
</service>
<reference cardinality="0..1" interface= "GoalVideo" policy="dynamic" target="(Agent=?Agent)"/>
<property name="self.osgi.precondition.LDAP" value="(light>30)"/>
</scr:component>
```
GoalLocalization(?Agent) : true ← {achieve(GoalLaser(?Agent)); ...}

<scr:component ... factory="LaserLocalization" name="CameraLocalization">
<implementation class= "LaserLocalization"/>
<property name="?Agent" type="String" value="The name of the robot to be localized">
<service>
  <provide interface="GoalLocalization"/>
</service>
<reference cardinality="0..1" interface="GoalLaser" target="(Agent=?Agent)/">
</scr:component>

Post-Conditions: The post-conditions of both component plans are specified with the OSGi XML service element. The Navigation component plan implements a move-to navigation behaviour in order to provide the service goal GoalNavigation, while both the CameraLocalization and the LaserLocalization component plans implement localization methods in order to provide localization updates through the service goal GoalLocalization, that is:

class Location {
  double x;
  double y;
}

interface GoalNavigation {
  void beAt (Location location);
}

interface GoalLocalization {
  Location getLocation();
}

Service Goal Requirements: Service goal requirements are declared using OSGi XML reference elements. The definition of Navigator declares its requirement of localization information as dynamic, in order to allow OSGi to activate it even when the reference to the Localization service goal is not resolved, thus avoiding to having to commit to a specific localization mechanism before the behaviour is started.

Noticeably, the definition of CameraLocalization includes Self-OSGi-specific property fields, self.osgi.precondition.LDAP, whose value may be used to characterise the context when the component plan is applicable. In the example, the LDAP pre-condition describes how CameraLocalization can only be used when the intensity of the ambient light, e.g. sensed by a light sensor component, is believed to be above a given threshold.

Variables: In order to link post-conditions with pre-conditions and service goal requirements, Self-OSGi allows the use of variable attributes whose name starts with the special character "?". Variables may be used as names of the property associated to a component plan in order to specify that the component plan can be instantiated with any value for that property. In such a case, the value of the variable is used as default value of the property. For instance, both the Navigator and the CameraLocalization component plans declare the property Agent to specify that they can be used by any agent to achieve, respectively, the GoalNavigation and the GoalLocalization service goals. Once the respective
component plans have been instantiated for a specific robot agent, i.e. Turtle-Bot, the services they provide will have an Agent attribute with value TurtleBot. However, in order for the same services to work, they must receive updates, respectively, of location and video data related to the same robot agent. Both XML descriptions specify this dependency by repeating the attribute ?Agent in the reference elements describing their required service goals. It is the responsibility of Self-OSGi to propagate the value of these variable properties from the post-condition/service side to the requirement/reference side, for instance, to wire a Navigator component activated in the TurtleBot robot agent, with a LaserLocalization activated for the same robot agent, rather than using the pure syntactic match (which could be satisfied by any localization data, e.g. related to other robots or used to represent the location of a human user).

3.3 Core Implementation:

The interested reader is referred to Dragone [3], for more detailed information on the internal architecture of each Self-OSGi platform.

The main difference from traditional agent platforms, such as the Agent Factory (AF) platform developed in UCD [18], is that agent container functionalities are built directly over the OSGi bundle and component container. In addition, rather than employing logic-based agent languages, Self-OSGi’s goals and plans are directly specified in Java, as discussed in the previous sections.

As a way of example, the following code is used to send a robot to a given location by initializing a standard OSGi ServiceTracker object to request the GoalNavigation service goal, before invoking it by passing the location coordinates. The special attribute selfosgi=true is used to demand the Self-OSGi management of the call. Noticeably, no other modifications are required to standard OSGi programming.

```java
ServiceTracker tracker = new ServiceTracker(...,
context.createFilter("(&(objectClass=\"GoalNavigation.class.getName\") (selfosgi=true))").open();
(GoalNavigation)(tracker.waitForService(0)).beAt (100, 200);
```

The service goal request is intercepted by Self-OSGi, which queries the OSGi DS for the list of all the components able to provide the requested service (i.e. LaserLocalization and CameraLocalization in the example). After that, Self-OSGi implements the BDI cycle by (i) finding all the component plans (installed in the same OSGi platform) with satisfied pre-conditions (i.e. which hold against the current content of the belief set), and (ii) instantiating (loading) and activating the most suitable one by using user-provided ranking components. Finally, Self-OSGi installs a proxy between the client that has originally requested a service goal, and the component activated to provide it. It is thanks to this mediation, that Self-OSGi can catch failures in the instantiated services activation, and trigger the selection of alternative component plans.

In the localization example, these features are used to make the robot reach its destination while opportunistically exploiting any suitable localization mechanism, for instance, starting with the CameraLocalization and then switching to
the LaserLocalization if the first fails when the ambient light drops below the given threshold.

4 Distributed Self-OSGi & PEIS Integration

While the Self-OSGi system described in [3] [4] and summarised in the previous section was limited to components and services running on a single platform and single JVM, the latest Self-OSGi version described in this paper has been fitted with extension mechanisms in order to provide seamless system distribution.

By default, such mechanisms leverage the R-OSGi distributed extension of OSGi to support service goal invocation across remote platforms. The other distributed extension of OSGi, D-OSGi, specifically targets Web Services technology and poses a much bigger overhead - 10MB - compared to the 230KB of R-OSGi footprint. R-OSGi can be deployed in minimal OSGi implementations and uses a very efficient network protocol. This makes it ideal for small and embedded devices with limited memory and network bandwidth.

Both R-OSGi and D-OSGi allows programs to bind and invoke remote services through port and proxy mechanisms. To this end, application components must register their services for remote access. Subsequently, other peers must connect to the R-OSGi registry of their peers to get access to their remote services.

Within Self-OSGi, the framework automatically manages these steps so that distribution becomes totally transparent to the application developer. In addition, Self-OSGi extends its automatic instantiation and selection of component plans to multiple platforms by integrating agent-based negotiation protocols within the standalone Self-OSGi system.

In the example depicted in Figure 2, a model bundle containing the specifications of three service goals (their Java interfaces) is equally shared by four platforms (A, B, C, D). However, the implementation of these service goals is distributed across the system. Specifically, platform A hosts two component plans implementing two of the service goals while platform D hosts one implementation of the service goal missing in platform A.

Developers can install a number of protocol components to handle the distribution of service goal requests. Thanks to OSGi DS, the fitting of a distribution protocol is done completely transparently from the application components, which do not need to be aware of their distribution across platforms.

Distribution protocols can range from simple delegation mechanism, which routes the service goal requests to specific platforms in the network, to more complex agent-style negotiation protocols, such as the contract net protocol (CNP) depicted in Figure 2. The latter can be used to query a group of platforms - which can be discovered via any network discovery solution - through a call for proposal (CFP) message. Upon reception of such a message, each platform will lookup their XML component repository and - in the case they can satisfy the request - reply with a bid message reporting the details of the component plan they consider most suitable to satisfy it. At this point, the original requester (platform
A in the example) can evaluate all bids, including its own, before sending a message (to platform D in the example) to accept the best bid. Upon reception of the accept message, the chosen platform will instantiate the selected component plan and automatically register with R-OSGi the service goal it provides. The initiating platform will then retrieve the reference to the service goal from the remote platform, create a proxy, and export it to the local OSGi registry.

![Diagram of automatic distribution of Self-OSGi Systems](image_url)

**Fig. 2.** Automatic distribution of Self-OSGi Systems

Within the RUBICON Control Architecture, the distributed extension of Self-OSGi has the following roles:

- provide the backbone upon which AI techniques, such as planning, cognitive reasoning and learning, can be integrated into a single system.
- provide a semantic vocabulary for expressing the capabilities and the requirements of all available devices and software components in the ecology, in order to support self-organization capabilities and the modular specification of the behaviour of the robotic ecology.
- enhance system’s scalability by framing it as a multi-agent system in order to leverage agent communication languages (ACLs) and multi-agent system (MAS) coordination & negotiation protocols.
- reduce the gap between the mainstream software solutions traditionally used in AmI/AAL domains and the state of the art techniques used in agent-oriented software engineering and in the control of robotic ecologies.

The final element to allow the use of the same mechanisms for the coordination of robotic ecologies is the interface between Self-OSGi and the PEIS middleware discussed in Section 2.
Specifically, the PEIS tuplespace is used to:

1. serve as platform discovery mechanism, by leveraging its peer-to-peer communication functionalities to re-create on each platform the directory of all the platforms available over the network. Self-OSGi uses these directories to find out the url of remote instances of Self-OSGi, before using R-OSGi to contact them by sending service goal requests and conducting auction-based negotiation protocols.

2. communicate with the PEIS-Init components installed on each platform, in order to manage PEIS components running outside the JVM, such as robotic behaviour components implemented in native C/C++ languages. In these cases, component plans in Self-OSGi acts as proxies of the underlying PEIS component they manage. A basic implementation - PEIS Component Monitor - provides generic functionalities to start/stop/configure peis components by setting the value of special tuples used to propagate these instructions via the tuplespace.

5 Testing Tools and Example

For the purpose of testing and demonstrating the overall implementation of the multi-agent framework presented in this paper, we have developed a set of PEIS tools capable of simulating the sensing and actuation aspects of components, while running the full framework for specification, introspection, deployment, communication and configuration. These tools are intended to simplify the development and debugging of the agent based tools and their interaction with the robotic ecology, and facilitate their adoption in real world situations.

The test illustrated in this section shows the ability of our framework to automatically generate a sequence of configurations to perform a given task in the current context (state). The test has been performed in a low-fidelity simulator emulating the behaviour of real devices distributed over three platforms:

1. a robot (robot-1) equipped with a laser ranging sensor, an odometry sensors, and a navigation component plan.
2. a robot (robot-2) equipped with a 3D ranging camera, an odometry sensors, a navigation and a tracking component plan.
3. a server (server) with an installation of a simultaneous localization and mapping (SLAM) component plan.

**Scenario:** In order to drive the robots, both navigation component plans must rely on odometry and localization information (their service goal requirement). However, none of the robot platforms have enough computational power to run a localization component plan locally. Fortunately, an instance of the SLAM component plan - running on the server - can provide location updates, as long as it receives data from both one ranging sensor and from the odometry of the robot. In addition, robot-2 can use its 3D ranging camera to observe the
scene from an external point of view, and compute its position relative to robot-1. Robot-2 can use this information and the knowledge of its own location, to provide an estimate of the absolute location of robot-1, which may then be used by robot-1 as an alternative to the location updates sent by the server.

These two alternative configurations, illustrated in Figure 3, can be easily deduced from the following dependencies (expressed in BDI-style notation here for simplicity in place of their actual XML form) between the localization, odometry, navigation and tracking service goals. The definition of these dependencies, i.e. the set of service goals and component plans, is packaged into a shared OSGi bundle that is installed on all the platforms of the ecology, while the XML files specify which component plan implementation is actually available to which platform.

**SERVER:**

\[
\text{GoalNavigation(?Agent) : true} \leftarrow \{ \text{achieve(GoalLocalization(?Agent)&GoalOdometry(?Agent))} \}\]

**ROBOT-1:**

\[
\text{GoalLocalization(?Agent) : true} \leftarrow \{ \text{achieve(GoalRanging(?Agent))} \} \\
\text{GoalRanging(?Agent) : true} \leftarrow \{ \text{achieve(GoalLaser(?Agent))} \} \\
\]

**ROBOT-2:**

\[
\text{GoalLocalization(?Agent) : true} \leftarrow \{ \text{achieve(GoalRanging(?Agent))} \} \\
\text{GoalRanging(?Agent) : true} \leftarrow \{ \text{achieve(GoalKinect(?Agent))} \} \\
\text{GoalLocalization(?Agent1) : true} \leftarrow \{ \text{achieve(GoalKinect(?Agent2)&GoalLocalization(?Agent2))} \} \\
\]

Finally, figure 4 and 5 show the outputs of the PEIS visualization tools, respectively showing the timeline and the connectivity graph of the wiring of the component plans in the ecology during the different phases of the experiment. In particular, the diagrams illustrate the status of the robotic ecology (1) after robot-1 was tasked to move to a given location, (2) after its laser was (artificially) disconnected to simulate a component failure, and (3) after the system had recovered from failure. The whole process can be summarised with the following steps:
1. robot-1 issues a Localization(robot-1) service goal request, before assigning it to server, which is randomly preferred to the robot-2 (the other bidder, as the post-condition of its tracking component plan also matches with the requested service goal).

2. server instantiates a SLAM component plan and issues two service goal requests to satisfy its service goal requirements, respectively for Ranging(robot-1) and Odometry(robot-1), before assigning both of them to robot-1 (the only bidder).

3. upon failure of the robot-1’s laser, the SLAM component plan on server also fails and robot-1 re-issues a Localization(robot-1) request. However, since the last time the SLAM component plan has failed, this time the service goal is assigned to robot-2.

4. robot-2 instantiates a tracking(robot-1) component plan, causing the issuing of requests for Localization(robot-2) and the successive wiring between its sensors and a new SLAM component plan on the server.

Fig. 4. PEIS Timeline Visualization Tool
6 Conclusions and future work

This paper has examined pre-existing middleware employed for the control of robotic ecologies and illustrated their use in conjuction with the Self-OSGi agent framework. The resulting framework provides re-usable, lightweight, modular and extensible mechanisms for the specification and the development of decentralized coordination mechanisms for robotic ecologies.

While the adoption of a multi-agent approach is often adopted for robot system design, a key and original result of our approach is that both system’s distribution and adaptation become orthogonal concerns freeing the developers to tackle application requirements. Once application components are implemented and their semantic and inter-dependencies described with Self-OSGi XML files, they can be freely distributed over the network, as the distributed Self-OSGi described in this paper will automatically manage their instantiation and configuration to achieve application’s objectives.

Future work will test the framework with larger scale problems and also seek to adapt agent/planning integration and agent learning techniques to tackle some of the main limitations of our architecture, such as its lack of look-ahead and its reliance on hard-coded pre-conditions of component plans.
ACKNOWLEDGMENT

This work has been supported by the EU FP7 RUBICON project (contract n. 269914).

References

1. G. Amato, e.a.: Robotic ubiquitous cognitive network. In: 3rd International Symposium on Ambient Intelligence, Salamanca, Spain (2012)
Evaluation of a Conversation Management Toolkit for Multi Agent Programming

David Lillis, Rem W. Collier, and Howell R. Jordan

School of Computer Science and Informatics
University College Dublin
{david.lillis,rem.collier}@ucd.ie
howell.jordan@lero.ie

Abstract. The Agent Conversation Reasoning Engine (ACRE) is intended to aid agent developers with the management of conversations to improve the management and reliability of agent communication. To evaluate its effectiveness, a problem was presented to two groups of undergraduate students, one of which was required to create a solution using the features of ACRE and one without.

This paper describes the requirements that the evaluation scenario was intended to meet and how these motivated the design of the problem that was presented to the subjects. The solutions were analysed using a combination of simple objective metrics and subjective analysis, which indicated a number of benefits of using ACRE. In particular, subjective analysis suggested that ACRE by defaults prevents some common problems arising that would limit the reliability and extensibility of conversation-handling code.

1 Introduction

The Agent Conversation Reasoning Engine (ACRE) is a suite of components and tools to aid the developers of agent oriented software systems to handle inter-agent communication in a more structured and reliable manner [1]. To date, ACRE has been integrated into the Agent Factory multi-agent framework [2] and is available for use with any of the agent programming languages supported by Agent Factory’s Common Language Framework [3]. Related work is presented in Section 2, followed by an overview of ACRE itself in Section 3.

The principal focus of this paper is to describe an experiment that was conducted to evaluate the benefits that ACRE can provide in a communication-heavy MAS. Section 4 outlines the motivations underpinning the design of the experiment and discusses how a scenario was designed with these in mind. In particular, we identified a number of variables that should be eliminated so as to ensure that comparisons of ACRE and non-ACRE code would be fair. This scenario was given to a class of undergraduate students, who were required to develop agents that could interact with a number of provided agents in order to trade virtual stocks and properties for profit. The subjects were divided into two groups: one using ACRE and one working without.
The code was analysed both objectively and subjectively and comparisons were made between code written using ACRE and code written without ACRE. This analysis is presented in Section 5. We then examined some conversation handling code written for the Jason multi agent platform to draw some conclusions about the wider applicability of this work in Section 6. Conclusions and ideas for future work are contained in Section 7.

2 Related Work

Although many well-known agent frameworks and languages have support for some Agent Communication Language (ACL), less attention has been paid to the notion of conversations between agents, where two or more agents will exchange multiple messages that are related to the same topic or subject, following a pre-defined protocol. The JADE toolkit provides specific implementations of a number of the FIPA interaction protocols [4]. It also provides a Finite State Machine (FSM) behaviour to allow custom interaction protocols to be defined in terms of how they are handled by the agents. Jason includes native support for communicative acts, but does not provide specific tools for the development of agent conversations using interaction protocols. A similar level of support has previously been present within the Agent Factory framework prior to the adoption of ACRE.

Agent toolkits with support for conversations include COOL [5], Jackal [6] and KaOS [7]. Other than FSMs, alternative representations for protocols include Coloured Petri Nets [8] and Dooley Graphs [9].

The comparative evaluation of programming toolkits, paradigms and languages is a matter of some debate within the software engineering community. One popular approach is to divide subjects into two groups with each asked to perform the same task [10–12]. To the greatest extent possible, objective quantitative measures are used to draw comparisons between the two groups. A common concept to evaluate is that of programmer effort, which has been measured in numerous different ways including development time [10, 11], non-comment line count [11] and non-commented source code statements [13]. These measures are used to ensure that a new approach does not result in a greater workload being placed on developers using it.

3 ACRE

ACRE is a framework that aims to give agent programmers greater control over the management of conversations within their MASs. It is motivated by the fact that many widely-used AOP languages/frameworks require communication to be handled on a message-by-message basis, with no explicit link between related messages. It frequently tends to be left as an exercise for the developer.
to ensure that messages that form part of the same conversation\(^1\) are interpreted as such. This section provides a brief overview of ACRE’s capabilities. For further information, it is presented in some detail in [1].

The principal components of ACRE are as follows:

- **Protocol Manager (PM):** A component that is shared amongst all agents residing on the same agent platform, the PM accesses online protocol repositories and downloads protocol definitions at the request of agents.

- **Conversation Manager (CM):** Each agent has its own CM, which is responsible for monitoring all its communication so as to group individual messages into conversations. Messages are compared to known protocol definitions and existing conversations to ensure that they are consistent with the available descriptions of how communication should be structured.

- **ACRE/Agent Interface (AAI):** Unlike the PM and CM, the AAI is not platform-independent, with its implementation dependent on the framework and/or AOP language being employed. The AAI serves as the API to the CM and PM: agents can perceive and act upon the ACRE components.

Additionally, the wider ACRE ecosystem provides an XML format for defining custom interaction protocols, a standard for the organisation of online protocol repositories and a suite of tools to aid developers in communication handling. These tools include a graphical protocol designer, a runtime conversation debugger, a GUI to manage and explore protocol repositories and a runtime protocol view to show what protocols have been loaded on an agent platform.

ACRE’s representation of protocols uses FSMs where transitions between states are activated by the sending and receiving of messages. An example of this can be seen in Figure 3. Here, the conversation is begun by a message being sent that matches any of the transitions emanating from the initial “Start” state. When this occurs, the variables (prefixed by the ? character) are bound to the values contained in the message itself. This will, for example, fix the name of the conversation initiator (bound to the ?player variable) and the other participant (?broker) so that subsequent messages must be sent to and from those agents.

If a message fails to match the specification of the relevant protocol, the CM will raise an error to make the agent aware of this. This feature is not readily available in existing AOP languages, as these typically depend on event triggers to positively match an expected message, with unrecognised or malformed messages being silently ignored if they match no rule.

From an AOP developer’s point-of-view, ACRE facilitates the implementation of interaction protocols by automatically checking messages against known protocols and conversations, providing information about the state of conversations as well as making available a set of actions that operate on them. The information available includes the participants in a conversation, the conversation state, the messages that make up the conversation history and the protocol.

\(^1\) In this work we draw a distinction between **protocols**, which define how a series of related messages should be structured and **conversations**, which are individual instances of agents following a protocol.
a conversation is following. Additionally, the agent is made aware of conversation events such as the conversation being advanced by a new message, cancellation, timeouts and errors (such as messages failing to match a transition in a defined protocol). Available actions include creating or advancing a conversation, cancelling existing conversations or communicating protocol errors to other participants.

\[
\text{+initialized : true <-}
\text{acre.init,}
\text{acre.addContact(agentID(banker,addresses("local:localhost"))),}
\text{acre.start(open,banker,request,openAccount)};
\]

\[
\text{+conversationAdvanced(?cid,done,?l) :}
\text{conversationProtocolName(?cid,open) <-}
\text{+bankAccount;}
\]

Fig. 1. ACRE code

Figures 1 and 2 show short code samples, written in AF-AgentSpeak (an adaptation of Rao’s AgentSpeak(L) [14] that is inspired by Jason [15]), showing how a simple interaction could be carried out with and without ACRE respectively. From these samples, the difference in approach may be seen.

The ACRE code adds the contact details of another agent only once, referring thereafter only to its name (“banker” in this example). Additionally, the events, beliefs and actions refer to details of a conversation (identified by a unique conversation identifier, ?cid). The second rule reacts to a conversationAdvanced event, which indicates that a conversation has been advanced by the communication of a message to a state named “done”. This state name is taken from the definition of the protocol that the conversation is following. The variables ?cid and ?l will be matched to the unique identifier of the conversation and the length of the conversation respectively. The conversation length is included to ensure that different messages that result in the same state (where the protocol contains a loop) will produce distinct events.

\[
\text{+initialized : true <-}
\text{openingAccount(banker),}
\text{.send(request,agentID(banker,addresses("local:localhost")),openAccount)};
\]

\[
\text{+message(inform,agentID(?sender,?addr), openedAccount(?id,?amt)) :}
\text{openingAccount(?sender) <-}
\text{+bankAccount;}
\]

Fig. 2. Non-ACRE code
Without ACRE, messages are dealt with individually, with the handling of a conversation being left to the developer (in this example, the adoption of the openingAccount(banker) belief records that the Player is engaging with the Banker agent to open a bank account). In addition, the name and address of the other agent must be provided each time a message is sent, and matched for every incoming message.

4 Evaluation Experiment

In order to evaluate the usefulness of ACRE, an experiment was designed whereby two groups of participants would write AOP code to tackle a particular problem. One group was required to write their code using the capabilities of ACRE whereas the other worked without ACRE. Before describing the experiment that was conducted, it is important to outline the motivations involved in its creation.

4.1 Motivations

The design of the problem was guided by the desire for the scenario to be communication-focused, accessible and reproducible, with a clearly defined implementation sequence and a clear reward for active agents, so that it is not possible for an agent to benefit by being inactive. Some of these motivations are specific to this particular experiment but many are desirable properties of any experiment where programming languages, paradigms or tools are being evaluated comparatively. These motivations are discussed in more detail below.

– **Focus on Communication:** The aim of the evaluation was to engage in a scenario that was communication-heavy. To facilitate this, it was decided that the problem should require developers to create a single agent, so that they would not be distracted from the core focus by having to deal with issues such as co-ordination and co-operation. A group of “core agents” would be provided with accompanying protocols, with which the participant’s agent must interact. Communicating with the core agents should be required from the beginning, with no progress being possible without communication.

– **Accessibility:** As the primary focus of the scenario is communication, little complex reasoning should be required to build a basic implementation that can perform well.

– **Reproducibility:** Every non-deterministic environment state change and core agent decision should be recorded, so that the experiment can be exactly replicated. Given a deterministic player agent, each replication should yield identical results.

– **Clearly defined goal:** From the point of view of participants, the assigned tasks, time allowed and scoring criteria should be clear.

– **Clearly defined implementation sequence:** Participants should not be able to gain a better score merely by implementing parts of their solution in a different order. The easiest way to ensure this is to fix the task order. In
the context of a communication-heavy experiment, task ordering may be enforced by making later protocols dependent on others, thus avoiding the need to monitor participants directly.

- **Rewards for Active Agents:** There should be no features of the experiment where an agent implementing that feature is at a disadvantage when compared to an agent that does not implement it.

### 4.2 Scenario

The scenario chosen for the experiment was a simple asset trading game. Each participant was required to develop an agent named *Player*. This agent was required to interact with a number of provided core agents in order to buy and sell virtual stocks and properties so as to increase the amount of money they had. The core agents were as follows:

- **Banker:** The Banker agent maintains a Player’s bank account. The first task of each agent is to interact with the Banker to open an account, in which an initial amount of virtual currency is placed.
- **Stockbroker:** This agent is responsible for buying and selling stocks. Players earn money by buying from the Stockbroker and selling later at a profit.
- **Guru:** The Guru agent is aware of how the market operates and can provide tips on which stocks will rise quickly in price and which should be avoided.
- **Auctioneer:** Properties can be bought from the Auctioneer agent. The value of properties rises quickly so they offer a method of making greater profits than on the stock market alone.
- **Bidder:** Bidders will participate in auctions organised by the Player to buy properties that the Player has previously purchased from the Auctioneer.

A number of features of the game were created with the motivations outlined above in mind. The *focus on communication* is maintained by having a number of protocols defined that the core agents are capable of following. Each protocol is based on one or more of the FIPA interaction protocols. These are summarised in Table 1. An illustration of one of these protocols is shown in Figure 3, namely the protocol invoked to buy stock from the Stockbroker. Similar illustrations were available for all protocols.

The *clearly defined goal* of the task is maximise capital. Thus participants are aware of what they need to do in order to be successful.

As a result of the *accessibility* motivation, the scenario is designed so that a Player can be successful using a simple strategy, without a great deal of advanced reasoning. This is particularly important when experiment participants are time-constrained (see Section 5).

By default, the movement of stock prices is determined randomly. An internal clock is used to track the time of the experiment: stock prices may change on every “tick” of the clock. A *reproducible* experiment may be conducted by loading pre-prepared stock prices at the beginning of the experiment, thus ensuring that the price movements are repeated across multiple experiments.
The ordering of the core agents in the list above reflects the order in which they should be interacted with, thus creating a clearly defined implementation sequence. The tasks are designed so that successfully completing a task will be dependent on previous tasks being completed first. Although there are no technical restrictions on the order in which participants may choose to implement the tasks, these dependencies discourage them from writing their implementations in a different order. For example, selling items to bidders is impossible before interacting with the Auctioneer to buy properties. However, these cannot be bought prior to interacting with the stock market, as the minimum property price is deliberately set to be higher than the Player’s initial capital. Similarly, the advice of the Guru is useless unless an agent can use it when interacting with the Stockbroker, and it is impossible to buy or sell stocks without having previously opened an account with the banker.

To reward active agents, the stock price calculation mechanism is intentionally artificial, in that the price of stocks always rises. This rewards developers for implementing features, to the detriment of idle agents. If stock prices can fall, an
Table 1. Core Agent Protocols

<table>
<thead>
<tr>
<th>Agent</th>
<th>Protocol Based On</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banker</td>
<td>open request</td>
<td>Open a bank account</td>
</tr>
<tr>
<td></td>
<td>enquiry query</td>
<td>Query your bank balance</td>
</tr>
<tr>
<td>Stock Broker</td>
<td>listing query</td>
<td>Get a list of available stocks</td>
</tr>
<tr>
<td></td>
<td>price query</td>
<td>Query the price of a particular stock</td>
</tr>
<tr>
<td></td>
<td>portfolio query</td>
<td>Query details of stocks currently owned</td>
</tr>
<tr>
<td></td>
<td>buy request</td>
<td>Buy a quantity/value of a particular stock</td>
</tr>
<tr>
<td></td>
<td>sell request</td>
<td>Sell a quantity/value of a particular stock</td>
</tr>
<tr>
<td>Guru</td>
<td>subscribe</td>
<td>Subscribe to the guru agent’s stock tips</td>
</tr>
<tr>
<td>Auctioneer</td>
<td>subscribe,</td>
<td>Subscribe to details of new auctions and</td>
</tr>
<tr>
<td></td>
<td>english-auction</td>
<td>participate in these auctions.</td>
</tr>
<tr>
<td>Bidder</td>
<td>sell contract-net</td>
<td>Sell a property</td>
</tr>
</tbody>
</table>

agent that does not participate in the market may end with more money than one that has interacted with a Stockbroker but that has lost money in doing so.

5 Discussion

The experiment participants were a final year undergraduate class in Fudan University, Shanghai, China. The evaluation was conducted as part of a module on Agent Oriented Programming. None of the participants had previous experience in developing MASs or in using an AOP language.

For consistency, all participants were required to write their code in AF-AgentSpeak so as to run within the Agent Factory framework. This removes any bias associated with the use of different AOP languages or frameworks.

Students were allowed three hours in a supervised laboratory setting in which to create their solutions. The fixed time period allows quantitative comparisons to be done with regard to the number of protocols each student implemented. The choice of a supervised in-class test ensured that each student submitted their own work. Subjects were permitted to access lecture notes and refer to manuals and user guides relating to Agent Factory, AF-AgentSpeak and ACRE.

The participants were divided into two groups of equal size using a random assignment: one group was requested to implement their solution using the extensions provided by ACRE whereas the other group was requested to implement their solution using the existing Agent Factory message-passing capabilities. In preparation for the experiment, a practical session was conducted so the participants had the opportunity to gain familiarity with both forms of message handling. This occurred a week prior to the evaluation so as to afford the students sufficient time to get accustomed to agent communication. Previous practical sessions held as part of the module exposed the participants to other aspects of AF-AgentSpeak and AOP programming in general.

The class consisted of 46 students in total, therefore 23 were asked to implement each type of solution. One student from the non-ACRE group did not
attend the evaluation, leaving 45 submissions. Additionally one other student from the non-ACRE group instead submitted a solution that did use ACRE.

Of the remaining 21 students in the non-ACRE group, one submission was not included in this research as only one protocol had been attempted and this attempt had not been successful, leaving a total of 20 non-ACRE submissions.

24 submissions were received that had used ACRE. Again, one of these has not been included in this analysis, as the agent submitted did not successfully interact with any core agent. This left a total of 23 submissions using ACRE.

The submissions were evaluated using both objective and subjective measures. Initially, some simple objective measures were employed to measure programmer effort (following the examples outlined in Section 2). Following this, the implementations were examined to identify any issues that the implementations may have failed to address.

5.1 Objective Measures

The principal aim of ACRE is to help developers to deal with complex communication more easily. As such, it is important to ensure that the use of ACRE does not add to the effort required to develop MASs. Objective measures are required to attempt to quantify programmer effort. Two simple metrics were employed for this purpose: 1) the number of protocols implemented within a specified time period and 2) the number of non-comment lines of code per protocol.

The first of these can be used to compare the two participant groups in terms of the time taken to implement protocols. Because of the ordering of the tasks, participants are encouraged to implement the protocols in the same order as one another. For example, it is not productive for a participant to begin by implementing the complex auctioneer protocols while others are implementing the simple protocol required to open a bank account. This helps to prevent the metric being skewed by certain subjects being delayed by the order in which they chose to implement their system.

Because there is variation in the number of protocols successfully implemented by each participant, the count of code lines is averaged for the number of protocols implemented. Again, the clear implementation sequence means that this is to the greatest extent possible measuring like-with-like.

<table>
<thead>
<tr>
<th></th>
<th>Protocols Implemented</th>
<th>Lines per Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACRE</td>
<td>5.43</td>
<td>18.35</td>
</tr>
<tr>
<td>Non-ACRE</td>
<td>5.85</td>
<td>27.06</td>
</tr>
</tbody>
</table>

Table 2 shows the average number of protocols implemented by each group, along with the average number of non-comment lines of code present per protocol implemented. For the number of protocols implemented, the difference is not
statistically significant using an unpaired t-test for \( p = 0.05 \). There is, however, a statistically significant difference in lines per protocol using the same test.

It can be seen that the participants in the ACRE group implemented marginally fewer protocols on average within the three-hour period. However, as this difference is not significant, it indicates that the speed of development is comparable whether ACRE is being used or not. This suggests that ACRE does not impose a steep learning curve compared to traditional methods of conversation handling.

It is interesting that the ACRE implementations did tend to have significantly fewer lines of code per protocol. Although this is a somewhat crude metric, it does suggest that the automated conversation handling of ACRE does reduce the amount of code that is required in order to successfully implement protocols.

Although objective measures are desirable in any evaluation, it is important to take a subjective view of the code also. While no significant difference in the amount of programmer effort required was observed, these metrics do not capture the quality of the implementations. The next section presents subjective analysis of the code submitted that attempts to identify this quality.

5.2 Subjective Assessment

The adoption of ACRE will be most beneficial if it improves code quality. To gauge this, the code was examined and a number of issues were found to be prevalent in the non-ACRE code. These issues meant that the solutions that were submitted were very closely tied to the scenario as presented, and would have required much more additional work to be done for an extended MAS.

By its nature, AF-AgentSpeak reacts to the receipt of messages using rules that have a triggering event and a context. The triggering event is a some event that has occurred (i.e. a change in the belief base) whereas the context is a set of beliefs that should be present for the rule to be relevant. When writing non-ACRE code in AF-AgentSpeak, the event that triggers the rule is the receipt of an incoming message and the context consists of beliefs about the state of the conversation (checking the sender, ensuring that the message follows the correct preceding message etc.). This can be seen in Figure 2. For the issues that were identified, the context of the rules were not written in the best way possible.

Identification of Issues The particular issues that were identified are as follows. Words in parentheses are short descriptions that are used to refer to each issue in the ensuing analysis:

- No Checking of Message Senders (Sender): Incoming messages were matched only using their performative and content, with no checks in place to ensure they were sent by the correct agent.
- No Checking of Conversation Progress (Progress): As the messages followed particular protocols, they were exchanged in a clearly-defined sequence. Many programmers did not attempt to check the context of messages that were received, treating them as individual messages.
- **Hard-coded name checking (Name):** When the message sender was checked, it was frequently the case that the name of the sender was hard-coded.

- **Checking addresses (Address):** Agent’s addresses were also hard-coded.

While a solution that includes these issues is capable of successfully participating in the trading scenario, their presence means that additional effort would be required to adapt the solution for a more open agent system or if the scenario were to be extended with additional protocols and/or agents.

When an agent fails to check the identity of the sender, this may have unintended consequences. For example, a Player agent would normally react to a recommendation from the Guru to buy a particular stock by following that recommendation. In a more open MAS, a malicious agent may send recommendations that cause other players to buy stocks that will not perform well. All that would be required is to send a message with the same performative and content as the Player would expect from the Guru.

Similarly, a Player that does not record the state of conversations is also susceptible to exploitation. For example, the protocol for buying stock (shown in Figure 3) insists that the Player must have accepted a proposal to buy stock before the purchase proceeds. However, without a notion of conversation, a Player may be persuaded that it has bought some quantity of a stock without it being involved in the process. If this is combined with the message sender not being checked, an agent other than a Stockbroker may trick a Player into buying stocks it had no intention to buy. As an aside, it is interesting to note that those agents that did record the progress of a conversation tended to use the state names provided in the ACRE FSM diagrams, which suggests that even for developers who do not use ACRE (or lack ACRE support in their AOP platform of choice), the availability of protocols defined in this way is useful for visualising and reasoning about conversations.

The two other issues relate to the difficulty in re-using the code for an extended scenario where conversations are conducted with different agents and/or multiple platforms. Even where the sender of a message was checked, it was frequently the case that this was hard-coded into the context of every rule. As such, the code was capable of conducting a conversation only with an agent of a specific name. If the scenario were to be extended so that multiple agents were capable of engaging in the same protocols (e.g. two Stockbroker agents that handled a different set of stocks) then these rules would all require re-writing to allow for additional agents. Similarly, hard-coding the addresses in the context of each rule limits the code to single platform MASs. Adding agents on another platform will also require all rules to be rewritten.

Because ACRE’s Conversation Manager automatically performs checking of this type, such issues cannot arise within ACRE code. As illustrated in the code from Figure 1, the triggering event is typically that a conversation has advanced, with the context being used to check other details about the conversation. Conversation participants need only be named when the conversation is initiated.

Assuming a pre-existing conversation, a `conversationAdvanced` event can only be triggered by a message that has been sent by the existing participant.
to the conversation and has a performative and content that match the next expected message in the protocol. If a message is sent by a different agent, an `unmatchedMessage` event is raised to indicate that the message does not belong to any particular conversation. This means that it is not necessary to check the message sender for each rule relating to communication, as the event cannot be triggered by the wrong agent.\(^2\)

This automatic checking also guards against out-of-sequence messages. In the example of the buying protocol shown in Figure 3, the Conversation Manager insists that the messages be communicated in the specified sequence, so the Stockbroker cannot inform the Player of a successful purchase unless the Player has previously agreed to that purchase (again, an `unmatchedMessage` event would be triggered).

Thus the use of ACRE automatically protects against these issues, meaning that were the MAS be more open or the scenario extended, far less effort would be required to adapt the existing ACRE code to the altered circumstances. ACRE can be seen to prevent certain coding styles that would restrict the extensibility and reusability of communication-handling code. Although subjects were not explicitly instructed to create generic code with wider applicability, we believe that ACRE’s prevention, by default, of these type of problems is a strong argument in its favour.

**Prevalence of Issues** For each of the issues outlined in the previous section, it is necessary to measure how prevalent they are amongst the implementations that were submitted. Each implementation was given one of three classifications with regard to each of the four issues identified:

- *Not susceptible:* The issue was not present for any rule in the implementation.
- *Totally susceptible:* The issue in question was present in every rule where it was relevant.
- *Somewhat susceptible:* The issue was present for some relevant rules but not all. This ranges from those implementations where a check was omitted only once to those where the check was only performed for one rule.

In relation to hard-coded name and address checking, these issues could not be present in agents that were totally susceptible to the issue of checking message senders. For those agents that did not check message senders at all, it is not possible for these other issues to arise. For this reason, in the following analysis, the figures shown for these two issues are displayed as a percentage of those agents in which was possible for them to arise.

Table 3 shows the prevalence of the issues amongst the non-ACRE submissions. Figures in parentheses are the absolute number of subjects each percentage

\(^2\) ACRE does not protect against messages sent by an agent other than that identified in the message’s `sender` field. This type of secure communication is considered to be a task for the underlying Message Transport Service.
Table 3. Issues present in non-ACRE code

<table>
<thead>
<tr>
<th></th>
<th>Sender</th>
<th>Progress</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Totally Susceptible</td>
<td>40% (8)</td>
<td>55% (11)</td>
<td>67% (8)</td>
<td>25% (3)</td>
</tr>
<tr>
<td>Somewhat Susceptible</td>
<td>30% (6)</td>
<td>30% (6)</td>
<td>0% (0)</td>
<td>17% (2)</td>
</tr>
<tr>
<td>Not Susceptible</td>
<td>30% (6)</td>
<td>15% (3)</td>
<td>33% (4)</td>
<td>58% (7)</td>
</tr>
</tbody>
</table>

relates to. Agents that were totally susceptible to the Sender issue are not included in the calculations for Name or Address, as these are only based on the number of agents that attempted to check the message sender.

From these, it can be seen that the issues raised were widespread amongst non-ACRE developers. The hard-coding of agents’ addresses is the only issue that was found in less than half of relevant agents.

Over two thirds of agents would react to messages sent by the wrong agent at least some of the time, with 40% failing to ever check the identity of a message sender. Of those that did check, two thirds hard-coded the name of the sender, which would require every rule to be re-written if the scenario was to be altered.

Just three participants (15%) always checked that messages were in the correct order. On further analysis, all of these implementations were somewhat susceptible to the Sender issue, which meant that there were no submissions where no issues were found.

We believe that these findings provide a strong argument in favour of the use of a conversation-handling technology such as ACRE that provides automated conversation checking and exception handling facilities without adding to the overall effort a programmer must go to when programming MASs in which communication plays a big part.

6 Comparison with Jason

The issues identified above arose specifically within AOP code using one specific programming language (AF-AgentSpeak) on one agent platform (Agent Factory). To show that ACRE’s conversation-handling capabilities could have a wider benefit than this single configuration, it was necessary to perform further analysis. To this end, we sought to examine how conversations are handled in a different agent framework that also lacks built-in conversation and protocol management. For this to be effective, we required that some best-practice conversation handling code must be available, so that any issues identified would not be as a result of poor coding practice or a lack of familiarity with the full capabilities of the language or platform.

Jason is a MAS development platform that makes use of an extended version of AgentSpeak(L) as its AOP language. It supports inter-agent communication using KQML-like messages. However, it lacks built-in support for conversation handling and protocol definitions. Jason was chosen for this analysis for two
principal reasons. Firstly, it is a popular, well-known platform. Secondly, a book is available that was written by Jason’s developers that includes sample code for performing a variety of tasks, including inter-agent communication [15]. As this code is written by the platform’s developers, we assume that it represents recommended best-practice.

Figure 4 is an extract from an implementation of a contract net protocol for Jason [15, p. 134]. This implementation is provided by the developers of Jason to illustrate how an interaction protocol may be implemented for that platform. The extract shows a plan that makes up part of the agent that initiates and coordinates the contract net. It is intended to be used when all bids have been received, so as to find the winner (line 7) and create an intention to announce the result to all the participants (line 9).

```prolog
@lc1[atomic]
+!contract(CNPId) : cnp_state(CNPId,propose) <-
  ++cnp_state(CNPId,contract);
  .findall(offer(O,A),propose(CNPId,O)[source(A)],L);
  .print("Offers are ",L);
  L \== []; // constraint the plan execution to at least one offer
  .min(L,offer(WOf,WAg)); // sort offers, the first is the best
  .print("Winner is ",WAg," with ",WOf);
  !announce_result(CNPId,L,WAg);
  --cnp_state(Id,finished).
```

Fig. 4. Sample Jason rule forming part of an implementation for Contract Net protocol

In the same way as the trading game presented in this paper, the sample MAS in which this agent was designed to run consists of a fixed set of agents with a particular purpose. Specifically, all other agents in the system were intended as participants in the contract net. As such, no allowance is made in the code for proposals being received from agents that were not party to the initial call for proposals. This can be seen in line 4 of the extract, which creates a list of offers that have been received based on any proposal that has been received from any source. This is the same as the Sender issue identified above. Extending this code for a more open MAS would require modification of the code to check that agents sending proposals are expected to do so. Jason does provide a method named SocAcc (meaning “socially acceptable”) that can prevent some types of message being processed if they are sent by inappropriate agents. Although this could be used to prevent non-participating agents from sending proposals, it is not sufficiently fine-grained to prevent an agent that is a party to one contract net from sending a proposal relating to another.

Figure 4 also illustrates that a Jason agent could also be susceptible to the “Progress” issue. The cnp_state belief in this extract is used to track the state of the conversation. As the code in question is written by experts, this belief is
present in a number of plans so that the state of the contract net conversation is known at all times. However, as our evaluation has shown, less experienced programmers are more prone to omitting this type of checking.

Another issue arises in the choice of an identifier for the conversation (referred to as the $\text{CNPId}$ variable in the extract shown). As presented, this ID is manually specified in the original intention that triggers the start of the contract net (not shown). ACRE assigns IDs to conversations automatically, meaning that the programmer need not be concerned with this aspect of conversation handling.

From this analysis, we can see that in the absence of integrated conversation handling, AOP developers using Jason are also susceptible to the issues outlined above. We believe that the type of conversation handling ACRE provides would help to avoid these pitfalls and so aid the development of reliable, scalable protocol implementations.

7 Conclusions and Future Work

We have described an experiment whereby two groups of students were required to solve a communication-focused problem with and without the use of ACRE. Objective metrics indicated that ACRE can reduce the amount of code required to implement the protocols provided when compared to implementing protocols without ACRE.

On further subjective analysis, a number of issues arose with non-ACRE code. These would require substantial modification of the code if the scenario were extended by the addition of additional Player agents, duplicate core agents, similar protocols or malicious agents of any type. The issues observed cannot occur with the use of ACRE, as the automatic conversation management ensures that both message senders and sequence are checked without developer intervention.

We also analysed some best-practice conversation-handling code written for Jason and observed that the issues identified are applicable to that platform. We therefore suggest that a conversation management framework such as ACRE is generally desirable to aid the development of communication-heavy MASs.

The experiment described in this paper was carried out by a single group of undergraduate students. It is intended to repeat the experiment for additional student groups, as this becomes possible. In particular, it is intended to use the same scenario in an upcoming Agent Oriented Software Engineering module. This is part of a part-time Masters course and the students are experienced industry software developers, albeit not using agents. This will give an insight into a different type of developer to the undergraduate students studied previously.

References

Compact and Efficient Agent Messaging

Kai Jander and Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{jander | lamersd}@informatik.uni-hamburg.de

Abstract. Messages are considered to be a primary means of communication between agents in multi-agent systems. Since multi-agent systems are used for a wide variety of applications, this also includes applications like simulation and calculation of computer generated graphics which need to employ a large number of messages or very large messages to exchange data. In addition, other applications target hardware which is resource constrained by either bandwidth or processing capacity. As a result, these applications have different requirements regarding their messages. This paper proposes a number of useful properties for agent messages and evaluates them with regard to various types of applications. Based on this evaluation a message format for Jadex called Jadex Binary is proposed, which emphasizes properties that are not traditionally the focus of agent message formats and compared them to some well-known formats based on those properties.

1 Introduction

Multi-agent systems enable the development of scalable and highly dynamic applications, facilitating their deployment on infrastructure such as structurally or spatially distributed systems, and the integration of mobile devices in such systems. An important means for agents to coordinate within an multi-agent application are messages passed between them. This mechanism is one of the enabling factors for autonomous behavior of agents, which enables them to be protected from direct influence by the rest of the system and establish a measure of robustness.

Nevertheless, certain classes of applications deployed on such systems have special requirements which appear to be in conflict with the focus of traditional agent message formats. Examples of this type of applications include real-time audio and video communication, distributed simulation and real-time distributed computer generated image (CGI) animation.

Traditionally, these requirements have not been the focus of agent messaging, which tends to target other useful properties that can also be important in multi-agent applications. As a result, it would broaden the application scope of agent systems if they specifically supported the requirements of such applications by providing an alternative message format.
In the following section we will attempt to identify typical requirements for agent messages and distill some that are especially important to the aforementioned classes of applications. We will then introduce some typical message formats used in agent systems and attempt to identify which application requirements they attempt to fulfill. Finally, we will present a compact message format that caters to the special class of application with real-time and bandwidth-restricted sets of requirements and compare it to traditional agent message formats, demonstrating key advantages for this special set of applications.

2 Features of Agent Message Formats

Since multi-agent applications cover a large spectrum of potential applications, there is an equally large number of features associated with agent messages which are potentially useful for different classes of applications. In addition to different application classes, different points in the development cycle may also emphasize the importance of certain features over others. While there is a large number of arguably useful features, we propose the following six features which are commonly mentioned and requested for multi-agent applications:

- **Human Readability** allows humans to read messages with standard tools like text viewers without the help of decoders or other special tools.
- **Standard Conformance** requires messages to conform to a published message format standard or language standard, allowing interaction between systems conforming to those standards.
- **A Well-formed Structure** defines a valid form for messages, allowing the system to distinguish between valid and invalid messages.
- **Editability** goes beyond human readability by allowing users to edit and restructure messages using standard tools such as text editors.
- **Performance** describes the computational requirements to encode and decode messages.
- **Compactness** evaluates the size of encoded messages.

In order to evaluate these features, we propose an example set of four common classes of applications. While there are many more potential applications, these applications are very common and would benefit from the use of agent technology. The first type of applications are **real-time applications**, where latency is a primary concern. Examples of this type of application can be found in any real-time communication application such as voice or video conference systems. High latency is generally unacceptable in such applications and may severely inhibit their functionality.

The second type of applications are **cross-platform applications**. For example, Agentcities[1] allows the use of multiple agent platforms and multiple types of agents, requiring precise definitions and standards among them. Correct interpretation of messages from other agents or platforms is key for such applications.

Another common type of applications involve **enterprise backend applications**. These applications often run on application servers on high-performance
intranets. It is important for such applications to provide quick access to the services required by the business in order to maintain high productivity.

The final type of applications are **mobile applications**, where a large number of nodes in the application are physically mobile and are typically connected using wireless connections. This means that the nodes are often restricted in terms of computational resources and network bandwidth. Energy supply is a key factor, stipulating modest use of resources even when more would be available.

<table>
<thead>
<tr>
<th></th>
<th>Real-time Applications</th>
<th>Cross-platform Applications</th>
<th>Enterprise Backend Applications</th>
<th>Mobile Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Readability</td>
<td>low</td>
<td>medium</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Standard Conformance</td>
<td>low</td>
<td>high</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>Well-formed Structure</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Editable</td>
<td>low</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>Performance</td>
<td>high</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Compactness</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

**Fig. 1.** Importance of message format features for different types of applications

Figure 1 shows the application types and the importance of the message format features. Some application types such as real-time applications and mobile applications have similar feature importance profiles for different reasons. While latency requires prudent use of resources for real-time applications, it is the energy and physical restrictions that make it a necessity for mobile applications. For cross-platform application the ability to interpret messages is key, so a standard-conformant and well-formed message format takes precedence over compactness and performance. Enterprise backend applications are more mixed, in that while the intranet typically provides abundant bandwidth, the large number of requests still requires good performance.

While an agent may have the option to open raw connection to other agents, bypassing the platform messaging service and supplying its own encoding and protocol, this is usually not advisable for the following reasons: On the one hand, developing an efficient transfer protocol involves a non-trivial amount of effort. It would therefore ease development effort if the agent message layer could be used. On the other hand, the agent system may be running within a restricted environment. For example, enterprise applications typically run on servers where the communication is tightly controlled for both support and security reasons. As a result, an agent may not be allowed to make connections outside what is provided by the agent platform.

Furthermore, there is another aspect concerning application development. In practice, there is often a distinction between the development phase of an application and production use in a business. For example, during development, applications often include additional logging and debug code to identify faults, include the use of assertions to validate program invariants and use tests to validate functionality. During production use, these features are often omitted in favor of higher throughput or lower latency.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Development</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Readability</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Standard Conformance</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Well-formed Structure</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Editable</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Performance</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Compactness</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

Figure 2. Importance of message format features during development and production use

Figure 2 demonstrates this difference between the two stages with regard to agent message formats. During development the ability to easily read and modify messages supports the developer in finding protocol errors and other implementation errors. In addition, a well-formed structure allows the use of validation tools to ensure message correctness; however, this changes during production use. Good encoding and decoding performance and message compactness aids both system throughput and latency. During production use, this takes precedence over issues like message readability, since the development has completed and it is no longer necessary for humans to read agent messages.

The next section will take a look at common agent message formats that have been traditionally used by multi-agent systems and show how well they support the proposed message format features. This will show that there is potential for improvements for both performance and compactness if other features are less of a concern.

3 Related Work

Over time, multi-agent systems have used a variety of message formats. Early system used simple ad-hoc languages in string-based formats; however, this resulted in languages that were specific to the application and made it difficult for multi-agent systems to interact. As a result, languages were developed to allow interchange between agent applications and agent platforms. One early attempt at defining an agent language was the Knowledge Query and Manipulation Language (KQML) [2]. However, it was quickly recognized that a standard language is useful for allowing communication between different agent systems.

Accordingly, the Foundation of Intelligent Physical Agents (FIPA) proposed two standards, the FIPA Agent Communication Language (ACL) [3] for the message structure and a specific language for the message content called FIPA SL [4] with different levels of complexity reaching from FIPA SL0 to FIPA SL2, both of which are used in popular agent platforms such as Jade [5].

This distinction between structure and content is retained in later formats as well. For example, while the Jadex Agent Platform [6] only uses a single XML-based format called Jadex XML, it distinguishes between message and content encoding. However, it uses Jadex XML for encoding both the message and content. Since the bulk of the message for the types of applications being targeted tends to be the content, the focus of this paper will be content encoding.
Nevertheless, as demonstrated by Jadex XML, the same principles can be applied
to message encoding as well.

<table>
<thead>
<tr>
<th>Feature</th>
<th>FIPA SL</th>
<th>Java Serialization</th>
<th>Jadex XML</th>
<th>Jadex Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Readability</td>
<td>*</td>
<td>*</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Standard Conformance</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>Well-formed Structure</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>Editable</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>Performance</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Compactness</td>
<td>-</td>
<td>*</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Fig. 3. Feature support by different methods of agent content encoding

When considering the agent format features proposed in Section 2, it becomes clear that even though some features are well-supported, other features were not the focus for those formats (cf. Fig. 3). For example, FIPA SL as a text-based format is quite readable and editable by humans, provides a definition of a well-formed structure and is a standard for agent messaging with wide support for many agent platforms. Jadex XML on the other hand, while not being a widely-used standard, has a well-defined and openly accessible schema and allows a human user to easily read and edit agent messages. Nevertheless, neither compactness nor performance seem to be the focus for either language. This is likely due to compactness and performance being in conflict with other features. For example, a compact format tends to be hard for humans to read.

The Jade platform recognizes the need for compact messages with good performance in some applications. It supports these features by adding content objects to messages, instead of a string-based content, but discourages this approach for lack of standard conformance. Jade uses the Java language serialization feature to encode such messages; however, while this approach is fairly compact and certainly providing good performance, it has multiple drawbacks. First, for a number of reasons listed in the specification it only supports classes that explicitly declare to implement a marker interface. While it is trivial to add the interface to classes, the source code of classes used in legacy applications may be unavailable. In addition, some useful built-in classes like BufferedImage do not implement this marker interface and there is no way to easily retrofit the serialization system to support this class. Furthermore, there appear to be compatibility issues, requiring a versioning convention using a marker field and carefully monitoring of the Java Virtual Machines used by the system. Finally, as we will show, the compactness of the serialization format can be further improved upon, especially without an additional compression cycle.

In the next section we will introduce an agent message format for Jadex called Jadex Binary which focuses solely on the compactness and performance features. This message format will be an alternative to the default Jadex XML used by Jadex, which can be used by application that have a strong need for those two features and do not require feature better supported by Jadex XML.
will then evaluate this new format based on the performance and compactness features based on a comparison with other agent message formats.

4 Format Description

Since the primary goal of the Jadex Binary format is to emphasize the compactness and performance properties of the format, it uses binary instead of string-based encoding. The primary concern is the serialization of the objects representing the message, such as, but not exclusively, ontology objects. In addition, some techniques are employed to prefer the compact encoding of common cases of data over rare cases, providing some simple compression based on the meta-information available from the objects and typical use cases. The format is based on a set of techniques to encode primitive types which are then used to encode more complex data. The following subsections will start with the primitive types and then proceed to more complex types.

4.1 Variable-sized Integers

A key concept used in Jadex Binary are variable-sized integers. The goal is to encode unsigned integer values in a variable-sized format that encodes small values with less space than larger ones. The technique is based on the encoding technique of element IDs in the Extensible Binary Meta-Language (EBML)[8], which again is based on variable encoding scheme used for UTF-8[9].

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Format</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1#######</td>
<td>0 to 127</td>
</tr>
<tr>
<td>2</td>
<td>01######</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>001#####</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0001####</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. Examples of variable-sized integers and their value ranges

A variable-sized integer is byte-aligned and consists of at least one byte (cf. Fig. 4). The number of zero bits starting from the highest-order bits before the first one-valued bit denotes the number of additional bytes called extensions that belong to this variable integer. The rest of the byte is then used to encode the highest-order bits of the integer value, the extensions then provide the lower order bits of the value. This value is then shifted by a constant equal to the end of the previous value range plus one. This technique of storing integer value uses less space to encode small values at the expense of additional space of high values.

The next part will describe how the format encodes boolean values which can be used in to extend variable integers to support negative values. Furthermore, variable integers are also heavily used as identifiers during string encoding.
4.2 Boolean Values

At first glance, encoding boolean values appears to be trivial since it only requires storing a single bit. However, a data stream that is not byte-aligned requires a considerable amount of processing to shift and pad bits during encoding and decoding, impacting the performance property of the format. As a result, a byte-aligned format is preferable. The Java language solves this issue by simply using a full byte to encode a single boolean value, however, this approach wastes almost 88% of the available space, which is incompatible with a compact language format.

![Diagram](image)

**Fig. 5.** Encoding of boolean values in the message: The first boolean value writes a byte-sized bit field that is reused by the next seven values

As a result, multiple boolean values are packed into a single bit. This is accomplished by writing a full byte where the first boolean value is written and updating that byte whenever additional boolean values are added (cf. Fig. 5). When the byte is filled with eight boolean values, another byte is written to the stream when the ninth boolean value is written. While the update cycles require some additional overhead on the part of the encoder, by having to update an earlier part of the byte stream, it reduces overhead for the decoder. During decoding, the byte is read when its first boolean value is read and then buffered for later reads.

This approach enables efficient storage of boolean values. This can be combined with the variable integer encoding to provide support for signed variable integers by writing a boolean sign flag before writing the absolute value as a variable integer.

4.3 Strings

Since string values tend to occupy a large part of typical messages, string encoding is a key part to ensure compactness. When a string is written by the
encoder, it is first checked if the string is already known. If not, the string is assigned a unique numerical ID and added to the set of known strings called the string pool.

![String Encoding Diagram](image)

**Fig. 6.** When a string is occurred for the first time in a message, it is encoded in full and assigned a unique ID allowing later occurrences to be encoded by referencing the ID.

The encoder then uses variable integer encoding to write the ID to the stream (cf. Fig 6). The string is then encoded using UTF-8 and its encoded size is written to the stream as a variable integer, followed by the encoded string itself. If the string is already known by the encoder, the encoder simply writes its ID as a variable integer, avoiding duplicate storage of the string.

Since the number of unique strings in a message is usually less than 128, a single byte is sufficient to encode any following occurrence of a string using variable integer encoding. Furthermore, the size of strings tends to be short, generally less than 16511 bytes or even 127 bytes, especially if few characters are used outside the first 128 unicode characters is used, allowing the string size to be encoded in one or two bytes.

All strings share the same string pool, whether it is used to encode an actual string value of the object or if it is used for other internal purposes such as type encoding. This maximizes the chance of finding duplicate strings in the pool, reducing message size.

### 4.4 Other Primitives

Other primitive values consist of integer and floating point types byte, short, int, long, float and double. All of these values are simply translated into network byte order[10] and added to the message. The only exception are 32-bit integer values. In many cases, these values are used as a kind of default integer type. For example, the Java language treats all untyped integer literals as this type. This leads to a disproportionately large set of 32-bit integer values to consist of mostly small numbers.

As a result, we found it to be advantageous with regard to the compactness property to encode 32-bit int values as variable-sized integers in the message data. While this can lead to large values exceeding the 4 bytes occupied by a 32-bit integer value, for common values the size is actually lower than 4 bytes, providing an overall net advantage in terms of size.
4.5 Complex Objects

Complex objects are needed to encode messages containing objects derived from ontologies such as concepts and their relations. It is also needed to encode sub-objects such as attributes. Aside from certain special cases which are discussed in the following subsections, complex objects generally have a type or class and contain a number of fields that can either be primitives or other complex objects. For this reason they can be traversed recursively, encoding each sub-object it contains as a complex object. The encoder only needs to keep track of objects that have already been encountered in order to avoid reference loops (Object A containing Object B containing Object A) and encode object references instead.

![Complex Object Table]

Fig. 7. A complex object is encoded using its class name, the number of sub-objects and pairs of field names and encoded sub-objects

As a result, the format for complex objects can be straightforward (cf. Fig 7). It starts with the fully-qualified class name, defining the type of the object. This is written to the message data using the string writing technique described in Section 4.3. Since some sub-objects may not be defined (i.e. reference null), not all of the sub-objects need to be encoded. Therefore, the class name is followed by the number of encoded sub-objects. This number is written to the message data as a variable integer as described in Section 4.1. Then the sub-objects are written by first writing the name of the field in the object containing the sub-object, then recursively encoding the sub-object itself. During decoding, the decoder first reads the class name and instantiates an object of that class. It then reads the number of sub-objects and finishes by decoding the sub-objects themselves, adding them to the object fields using the appropriate accessor methods.

Generally, it is expected that the objects offer accessor methods as described in the Java Bean specification and the encoder will only encode fields for which such accessor methods are available. However, using annotations, a class may declare that the encoder should encode the field regardless of the existence of accessor methods. In this case the fields are accessed directly using the Java reflection API.

4.6 Arrays

Arrays are encoded in a manner similar to complex objects, starting with the array type and concatenating the number of array components and the compo-
ponents themselves. However, since the array type already provides type information about the array components, this information can be used to reduce the amount of information that needs to be stored in the message. The array encoding therefore supports two modes, a raw mode which is used for primitive type values and a complex mode for objects. In raw mode, the components are just written without additional information about the type of each component. This can only be done in case of primitive types for two reasons: First, primitive types cannot be subclassed, thus the array type is always sufficient to derive the component type. Second, primitive values are passed by value and do not have a null reference that requires special encoding.

This is not the case for objects. The array type may only refer to a superclass of the component object and the component object may simply be a null reference. This means that type information about each array component needs to be included. However, in most cases it is safe to assume that the array type describes the type of the component. Therefore, each component is preceded by a boolean value indicating whether further component type information is stored or if the type information can be derived from the array type. Only if this flag indicates that type information is stored, for example when dealing with subclasses or null values, the flag is followed by the type information. Otherwise, the component object is directly appended after the boolean flag.

In the next section we will evaluate Jadex Binary in terms of the performance and compactness features and compare it to three other message formats. This was done with a number of tests in which the message formats were measured and evaluated.

5 Evaluation

In order to evaluate the performance and compactness feature of Jadex Binary, we conducted a series of tests. The experiments were conducted using an Intel i5 750 processor with four cores clocked at 2.67 GHz. The machine was supplied with 8 GiB of memory; however, the Java heap size was limited to 2 GiB. The Java environment used was the Oracle Java SE 6 Update 31 which was running on a current version of Gentoo Linux compiled for the x86-64 instruction set with an unpatched Linux 3.2.2 kernel.

The content formats used were FIPA SL, the built-in Java serialization, Jadex XML and Jadex Binary. The Jade agent platform 4.1.1 using the BeanOntology was used as a representative of Java SL encoding. A test class representing an agent action was used as data set to be encoded. The agent action sample contained a 514 byte string literal, a second string containing a randomized long value encoded as string, a single integer value, an array of 20 integer literals, an array of boolean values and finally an array of objects of the class itself to represent recursively contained sub-objects.

For the compactness tests, this array contained 100 further instances of the class, which itself had the field set to null. For the performance tests, the number of objects in that array was varied, starting at 10000 objects and increasing in
steps of 10000 up to 100000 objects. The compactness was measured by counting the number of bytes of the encoded content. If the encoded content was a string, it was converted to a byte representation using UTF-8 encoding (other encodings like UTF-16 would have been possible but would have resulted in worse results). For the performance, the time between start and end of the encoding cycle was measured, other tasks like encoder and object setup were not considered.

In the following, we will present the results of both the performance and the compactness features. While the test data is certainly artificial, we have tried to supply what we think is a good cross-section of possible data cases.

5.1 Performance

In order to avoid interference with lazy initialization procedures and the just-in-time compilation of the Java VM, all performance tests were run twice, with the results of the first test run being discarded. This allowed the Java environment to compile the code and the encoding framing to initialize constants during the first pass so that the real performance figures could be obtained during the second pass.

![Graph](image)

**Fig. 8.** Results for the performance measurements, Jade FIPA SL encoding requiring a disproportionally long time

The result of the tests can be seen in Figure 8. While all encoding time measurements seem to increase linearly with the number of objects, the FIPA SL encoding provided by Jade appears to require an unusually long time. As expected, both Jadex Binary and the Java serialization mechanism provide substantially better results; however, even Jadex XML which is not intended to be
optimized for this format feature still offers substantially lower encoding times than Jade FIPA SL encoding.

Figure 9 provides a closer look at the three highest performing formats. While Jadex Binary clearly provides an advantage over Jadex XML by roughly a factor of two, the Java serialization is almost an order of magnitude faster. Initial analysis seems to suggest this is due to the use of the Java Reflection API used by both Jadex XML and Jadex Binary, which the Java serialization mechanism can avoid due to its built-in nature. However, Java serialization has further drawbacks as outlined in Section 3, meaning it is not a general solution to the problem and has a more narrow scope of environments in which it can be useful.

5.2 Compactness

In order to test for content compactness, the test object was passed to the encoder and the number of bytes of the encoded object was measured. Since the test object contained a fair amount of test data, the resulting content sizes were expected to be large.

As can be seen in Figure 10, the differences between the four formats are quite substantial. Jadex XML is barely half the size of the FIPA SL encoding and both Java serialization and Jadex Binary are substantially smaller still. In fact, Jadex Binary clearly provided the most compact representation of the test object, being smaller than even the Java serialization format by a factor of roughly 2.5.
A fair amount of this is likely to be due to redundant information, especially of string values, which Jadex Binary can exploit (though Jadex XML uses a similar mechanism). In addition, text-based formats like FIPA SL and Jadex XML use a large amount of redundant strings to represent their formatting, such as tags in the case of XML.

In order to test this assumption, another set of tests was performed, which were identical to the previous tests but added an additional compression pass, converting it to the gzip-format, which uses the DEFLATE algorithm[11] to reduce data redundancy.

The results shown in Figure 11 substantiate the assumption. The DEFLATE algorithm drastically reduced the redundancies in both FIPA SL and Jadex XML with FIPA SL now even coming out ahead of Jadex XML. Nevertheless, both
Java serialization and Jadex Binary still show an advantage in compactness with Jadex Binary maintaining a slim margin over Java serialization.

![Bar chart](image)

**Fig. 12.** An additional compression pass increases total encoding time

Since the compression pass substantially reduces the size of messages, especially for verbose formats, it may suggest that starting out with a compact format gives only a marginal advantage. However, data compression is not free in terms of computation time. While compression helps compactness, this issue has to be weighed against the performance message feature. Despite the DEFLATE algorithm being a comparably fast compression algorithm, Figure 12 shows that it adds a substantial amount to the total encoding time of the content. In fact, the additional time required seems to grow with the number of bytes in the uncompressed content, which is reasonable considering the algorithm must evaluate every byte of the uncompressed data at least once to produce a reversible output.

As a result, data compression does not appear to be generally beneficial when both performance and compactness are important; however, it is another useful tool to adjust the balance between the two language features. In the next section we will discuss further improvements, future work and provide a conclusion on the performance of Jadex Binary.

### 6 Future Work and Conclusion

The evaluation of Jadex Binary in Section 5 appears to provide sufficient evidence that Jadex Binary already has significant advantages in both compactness and performance. However, the performance results of the Java serialization shows that further performance improvements may be possible. One way of further
reducing the overhead of Jadex Binary is to reduce the use of the Java Reflection API to access complex objects. This could be accomplished by injecting bytecode-engineered delegate classes which use direct method calls to retrieve and set bean properties.

In addition, the encoder and decoder of Jadex Binary are largely independent of the Jadex platform. It would therefore be possible to include the message format in other agent platforms, thereby allowing them to offer an alternative compact message format for agent communications for certain types of applications.

Overall, Jadex Binary is both able to represent agent messages in a compact form and perform in a reasonably fast manner. Since these two features were the primary goal of Jadex Binary, it does so by sacrificing others like human readability. Nevertheless, if those features are important, other established languages already provide sufficient support. The addition of Jadex Binary allows a developer of a multi-agent system to pick the kind of format that provides the best match for the requirements of a specific application and switch the format depending on state of the application in the development cycle.

References


Query Caching in Agent Programming Languages

Natasha Alechina¹, Tristan Behrens², Koen Hindriks³, and Brian Logan¹

¹ School of Computer Science
University of Nottingham
Nottingham NG8 1BB UK

² Department of Informatics
Clausthal University of Technology
Germany

³ Delft University of Technology

Abstract. Agent programs are increasingly widely used for large scale, time critical applications. In developing such applications, the performance of the agent platform is a key concern. Many logic-based BDI-based agent programming languages rely on inferencing over some underlying knowledge representation. While this allows the development of flexible, declarative programs, repeated inferencing triggered by queries to the agent’s knowledge representation can result in poor performance. In this paper we present an approach to query caching for agent programming languages. Our approach is motivated by the observation that agents repeatedly perform queries against a database of beliefs and goals to select possible courses of action. Caching the results of previous queries (memoization) is therefore likely to be beneficial. We develop an abstract model of the performance of a logic-based BDI agent programming language. Using our model together with traces from typical agent programs, we quantify the possible performance improvements that can be achieved by memoization. Our results suggest that memoization has the potential to significantly increase the performance of logic-based agent platforms.

1 Introduction

Belief-Desire-Intention (BDI) based agent programming languages facilitate the development of rational agents specified in terms of beliefs, goals and plans. In the BDI paradigm, agents select a course of action that will achieve their goals given their beliefs. To select plans based on their goals and beliefs, many logic-based BDI-based agent programming languages rely on inferencing over some underlying knowledge representation. While this allows the development of flexible, declarative programs, repeated inferencing triggered by queries to the agent’s knowledge representation can result in poor performance. When developing multiagent applications for large scale, time critical applications such performance issues are often a key concern, potentially adversely impacting the adoption of BDI-based agent programming languages and platforms as an implementation technology.

In this paper we present an approach to query caching for agent programming languages. Our approach is motivated by the observation that agents repeatedly perform queries against a database of beliefs and goals to select possible courses of action.
Caching the results of previous queries (memoization) is therefore likely to be beneficial. Indeed caching as used in algorithms such as Rete [1] and TREAT [2] has been shown to be beneficial in a wide range of related AI applications, including cognitive agent architectures, e.g., [3], expert systems, e.g., [4], and reasoners, e.g., [5]. However that work has focused on the propagation of simple ground facts through a dependency network. In contrast, the key contribution of this paper is to investigate the potential of caching the results of arbitrary logical queries in improving the performance of agent programming languages. We develop an abstract model of the performance of a logic-based BDI agent programming language, defined in terms of the basic query and update operations that form the interface to the agent’s knowledge representation. Using our model together with traces from typical agent programs, we quantify the possible performance improvements that can be achieved by memoization. Our results suggest that memoization has the potential to significantly increase the performance of logic-based agent platforms.

The remainder of the paper is organised as follows. In Section 2 we introduce an abstract model of the interface to a logic-based BDI agent’s underlying knowledge representation and an associated performance model. Section 3 presents experimental results obtained from traces of typical agent programs and several key observations regarding query and update patterns in these programs. Section 4 introduces two models to exploit these observations and improve the efficiency of the use of Knowledge Representation Technologies (KRTs) by agent programs. Section 5 discusses related work, and Section 6 concludes the paper.

2 Abstract Performance Model

In this section, we present an abstract model of the performance of a logic-based agent programming language as a framework for our analysis. The model abstracts away details that are specific to particular agent programming languages (such as Jason [6], 2APL [7], and GOAL [8]), and focuses on key elements that are common to most, if not all, logic-based agent programming languages.

The interpreter of a logic-based BDI agent programming language repeatedly executes a ‘sense-plan-act’ cycle (often called a deliberation cycle [9] or agent reasoning cycle [6]). The details of the deliberation cycle vary from language to language, but in all cases it includes processing of events (sense), deciding on what to do next (plan), and executing one or more selected actions (act). In a logic-based agent programming language, the plan phase of the deliberation cycle is implemented by executing the set of rules comprising the agent’s program. The rule conditions consist of queries to be evaluated against the agent’s beliefs and goals (e.g., plan triggers in Jason, the heads of practical reasoning rules in 2APL) and the rule actions consist of actions or plans (sequences of actions) that may be performed by the agent in a situation where the rule condition holds. In the act phase, we can distinguish between two different kinds of actions. Query actions involve queries against the agent’s beliefs and goals and do not change the agent’s state. Update actions, on the other hand, are either actions that directly change the agent’s beliefs and goals (e.g., ‘mental notes’ in Jason, belief update...
actions in 2APL), or external actions that affect the agent’s environment, and which may indirectly change the agent’s beliefs and goals.

In a logic-based agent programming language, the agent’s database of beliefs and goals is maintained using some form of declarative knowledge representation technology. Queries in the conditions of rules and query actions give rise to queries performed against the knowledge representation. Update actions give rise (directly or indirectly) to updates to the beliefs and goals maintained by the knowledge representation. For example, Figure 1 illustrates example rules from Jason, 2APL and GOAL agent programs, which select a move action to move a block in the Blocks World environment. While the rules appear quite different and have different components, the evaluation of

\[ +!\text{on}(X,Y) \leftarrow !\text{clear}(X); !\text{clear}(Y); \text{move}(X,Y). \]

(a) Jason

\[ \text{allOnTable} \leftarrow \text{on}(X,Y) \text{ and } \text{clear}(X) \text{ and } \neg(Y=\text{table}) \mid \{ \text{@blocksworld(move}(X,\text{table}),,); \text{On}(X,\text{table}) \} \]

(b) 2APL

\[ \text{if } \text{a-goal}(\text{tower}([X| T])) \text{ then move}(X, \text{table}). \]

(c) GOAL

Fig. 1: Example Blocks World rules

the conditions of each rule gives rise to similar queries to the underlying knowledge representation. In this example, the terms on, clear and tower are predicates which are evaluated by querying the belief and goal bases of the agents. Similarly, the agent programs use logical rules (Horn clauses) to represent knowledge about the environment. For example, Figure 2 illustrates a rule used in the Jason Blocks World agent to determine whether a set of blocks constitutes a tower.

\[ \text{tower}([X]) \leftarrow \text{on}(X,\text{table}). \]
\[ \text{tower}([X,Y|T]) \leftarrow \text{on}(X,Y) \text{ and } \text{tower}([Y|T]). \]

Fig. 2: Example Jason logical rule

The 2APL and GOAL agents use the same recursive rule in Prolog format with ‘&’ replaced by ‘,’. Similarly, in each case, execution of external actions such as move and internal belief and goal update actions give rise to updates to the agent’s beliefs and goals, either indirectly through perception of the environment (in the case of external action) or directly (in the case of internal actions).

From the point of view of the agent’s knowledge representation, the three steps in the sense-plan-act cycle can therefore be mapped onto two kinds of knowledge representation functionality. The knowledge representation must provide functionality for
querying an agent’s beliefs and goals when applying rules or executing query actions in
the agent’s plans, and for updating an agent’s beliefs and goals upon receiving new in-
formation from other agents or the environment, or because of internal events that occur
in the agent itself. Our performance model therefore distinguishes two key knowledge
representation phases that are common to virtually all logic-based agent programming
languages: a query phase and an update phase. The two phases together constitute an
update cycle.

The model is illustrated in Figure 3. The query phase includes all queries processed
by the agent’s knowledge representation in evaluating rule conditions to select a plan
or plans, and in executing the next step of the agent’s plans (e.g., if the next step of
a plan is a belief or goal test action). The update phase includes all updates to the
agent’s knowledge representation resulting from the execution of the next step of a
plan, where this step changes the agent’s state directly (e.g., the generation of subgoals
or the addition or deletion of beliefs and goals), and updating the agent’s state with
new beliefs, goals, messages or events at the beginning of the next sense-plan-act cycle.
Note that update cycles do not necessarily correspond one-to-one to deliberation cycles.
For example, in Jason and 2APL the action(s) performed at the end of a deliberation
cycle may be internal actions (such as test actions) that do not update the agent’s beliefs
and goals, and in these languages the query phase may include queries from several
deliberation cycles. In what follows, we assume that the query phase occurs first and
the update phase second, but our results do not depend on this particular order and a
similar analysis can be performed if the order of the phases is reversed.

To develop our performance model in detail, we must first make the query/update
interface to the agent’s knowledge representation precise. Different agent program-
ing frameworks utilise different types of databases to store different parts of the
agent’s state. For example, most logic-based agent programming languages use dif-
ferent databases for beliefs and for goals, and almost all languages (with the exception
of GOAL) maintain bases that store plan-like structures or intentions. Here we focus
on those aspects common to most logic-based agent programming languages, namely
operations on the agent’s beliefs and goals, and abstract away from the details of their
realisation in a particular agent platform. In particular, we ignore operations performed
on databases of intentions or plans. Although agent platforms do perform operations on
intentions and plans that can be viewed as queries and updates, these operations vary widely from platform to platform and typically do not involve logical inference.

The first key KRT functionality is querying a database. A query assumes the presence of some inference engine to perform the query. In many agent platforms, a distinction is made between performing a query to obtain a single answer and to obtain all answers. In what follows, we abstract away from the details of particular inference engines provided by different agent platforms and represent queries by the (average) time required to perform the query. The second key functionality is that of modifying or updating the content of a database. With the exception of recent work on the semantic web and on theory progression in situation calculus, update has not been a major concern for classical (non-situated) reasoners. However it is essential for agents as they need to be able to represent a changing and dynamic environment. All the agent platforms we have investigated use a simple form of updating which involves simply adding or removing facts. In cases where the agent platform adopts the open world assumption, one needs to be slightly more general and support the addition and removal of literals (positive and negated facts).

Based on this model, we can derive an analysis of the average case performance for a single update cycle of an agent. Our analysis distinguishes between costs associated with the query phase and the update phase of an update cycle. We assume that the agent performs on average $N$ queries in the query phase of an update cycle. If the average cost of a query is $c_{qry}$, then the average total cost of the query phase is given by

$$N \cdot c_{qry}$$

In general, the same query may be performed several times in a given update cycle. (We provide support for the fact that queries are performed multiple times in a cycle below.) If the average number of unique queries performed in an update cycle is $K$, then on average each query is performed $n = N/K$ times per cycle.

The total average cost of the update phase of an update cycle can be derived similarly. In logic-based agent programming languages, updates are simple operations which only add or remove facts (literals) from a database, so it is reasonable to consider only the total number of updates when estimating the cost of the update phase. If $U$ is the average number of updates (i.e., adds and deletes) per cycle and $c_{upd}$ is the average cost of an update, then the average total cost of the update phase is given by

$$U \cdot c_{upd}$$

Combining both the query and update phase costs yields:

$$N \cdot c_{qry} + U \cdot c_{upd}$$

(1)

3 Experimental Analysis

To quantify typical values for the parameters in our abstract performance model, we performed a number of experiments using different agent platforms and agent and environment implementations. We stress that our aim was not to determine the absolute
or relative performance of each platform, but to estimate the relative average number of queries and updates performed by ‘typical’ agent programs, and their relative average costs on each platform, in order to determine to what extent caching may be useful as a general strategy for logic-based agent programming languages. To this end, we selected three well known agent platforms (Jason [6], 2APL [7] and GOAL [8]), and five existing agent programs/environments (Blocks World, Elevator Sim, Multi-Agent Programming Contest 2006 & 2011, and Wumpus World).

The agent platforms were chosen as representative of the current state of the art in logic-based agent programming languages, and span a range of implementation approaches. For example, both 2APL and GOAL use Prolog engines provided by third parties for knowledge representation and reasoning. 2APL uses the commercial Prolog engine JIProlog [10] implemented in Java, whereas GOAL uses the Java interface JPL to connect to the open source SWI-Prolog engine (v5.8) which is implemented in C [11]. In contrast, the logical language used in Jason is integrated into the platform and is implemented in Java.

The agent programs were chosen as representative of ‘typical’ agent applications, and span a wide range of task environments (from observable and static to partially observable and real-time), program complexity (measured in lines of code, LoC), and programming styles. The Blocks World is a classic environment in which blocks must be moved from an initial position to a goal state by means of a gripper. The Blocks World is a single agent, discrete, fully observable environment where the agent has full control. Elevator Sim is a dynamic, environment that simulates one or more elevators in a building with a variable number of floors (we used 25 floors) where the goal is to transport a pre-set number of people between floors [12]. Each elevator is controlled by an agent, and the simulator controls people that randomly appear, push call buttons, floor buttons, and enter and leave elevators upon arrival at floors. The environment is partially observable as elevators cannot see which buttons inside other elevators are pressed nor where these other elevators are located. In the 2006 Multi-Agent Programming Contest scenario (MAPC 2006) [13] teams of 5 agents explore grid-like terrain to find gold and transport it to a depot. In the 2011 Multi-Agent Programming Contest scenario (MAPC 2011) [14] teams of 10 agents explore ‘Mars’ and occupy valuable zones. Both MAPC environments are discrete, partially observable, real-time multi-agent environments, in which agent actions are not guaranteed to have their intended effect.

Finally, the Wumpus World is a discrete, partially observable environment in which a single agent must explore a grid to locate gold while avoiding being eaten by the Wumpus or trapped in a pit. For some of the environments we also varied the size of the problem instance the agent(s) have to deal with. In the Blocks World the number of blocks determines the problem size, and in the Elevator Sim an important parameter that determines the size of a problem instance is the number of people to be moved between floors. The size of problem instances that we have used can be found in the first column of Tables 2 through 6.

It is important to stress that, to avoid any bias due to agent design in our results, the programs were not written specially for the experiments. While our selection was therefore necessarily constrained by the availability of pre-existing code (in particular versions of each program were not available for all platforms), we believe our results are
representative of the query and update performance of a broad range of agent programs ‘in the wild’. Table 1 summarises the agents, environments and the agent platforms that were used in the experiments.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Agent Platform</th>
<th>LoC</th>
<th>Deliberation Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks World</td>
<td>Jason</td>
<td>34</td>
<td>104-961</td>
</tr>
<tr>
<td></td>
<td>2APL</td>
<td>64</td>
<td>186-1590</td>
</tr>
<tr>
<td></td>
<td>GOAL</td>
<td>42</td>
<td>16-144</td>
</tr>
<tr>
<td>Elevator Sim</td>
<td>2APL</td>
<td>367</td>
<td>3187-4010</td>
</tr>
<tr>
<td></td>
<td>GOAL</td>
<td>87</td>
<td>2292-5844</td>
</tr>
<tr>
<td>MAPC 2006</td>
<td>Jason</td>
<td>295</td>
<td>2664</td>
</tr>
<tr>
<td>MAPC 2011</td>
<td>GOAL</td>
<td>1588</td>
<td>30</td>
</tr>
<tr>
<td>Wumpus World</td>
<td>Jason</td>
<td>294</td>
<td>292-443</td>
</tr>
</tbody>
</table>

Table 1: Agents and Environments

3.1 Experimental Setup

To perform the experiments, we extended the logging functionality of the three agent platforms, and analysed the resulting query and update patterns in the execution traces for each agent/environment combination. The extended logging functionality captured all queries and updates delegated to the knowledge representation used by the agent platform and the cost of performing each query or update.

In the case of 2APL and GOAL, which use a third party Prolog engine, we recorded the cost of each query or update delegated to the respective Prolog engine. In these languages, Prolog is used to represent and reason with percepts, messages, knowledge, beliefs, and goals. Action preconditions and test goals are also evaluated using Prolog. Prolog queries and updates to the Prolog database therefore account for all costs involved in the query and update phases of an update cycle. In the case of Jason, the instrumentation is less straightforward, and involved modifying the JASON belief base to record the time required to query and update percepts, messages, knowledge and beliefs. The time required to process other types of Jason events, e.g., related to the intentions or plans of an agent, was not recorded.

We ran each of the agent/environment/platform combinations listed in Table 1 until the pattern of queries and updates stabilised (i.e., disregarding any ‘start up’ period when the agent(s), e.g., populate their initial representation of the environment). For different agent environments, this required different numbers of deliberation cycles (listed in the Deliberation Cycles column in Table 1). For example, fewer deliberation cycles are required in the Blocks World to complete a task than in other environments, whereas in the Elevator Sim environment thousands of deliberation cycles are required to reach steady state. For the real-time Multi-Agent Programming Contest cases, the simulations

---

4 In contrast to 2APL and GOAL, Jason does not have declarative goals.
were run for 1.5 minutes; 1.5 minutes is sufficient to collect a representative number of cycles while keeping the amount of data that needs to be analysed to manageable proportions. For each agent/environment/platform run, the time required to perform each query or update resulting from the execution of the agent’s program was logged, resulting in log files as illustrated in Figure 4. Here, add and del indicate updates, and

\[
\begin{align*}
\text{add} & \quad \text{on(b2,table)} & 30 \\
\text{add} & \quad \text{on(b9,b10)} & 43 \\
\text{del} & \quad \text{on(b4,b6)} & 21 \\
\text{query} & \quad \text{tower(’.’ (X,T))} & 101 \\
\text{query} & \quad \text{tower(’.’ (b7,[]))} & 51 \\
\end{align*}
\]

Fig. 4: Example log file

query indicates a belief or goal query, followed by the updated belief or goal, or query performed, and the time required in microseconds.

3.2 Experimental Results

In this section we briefly present the results of our analysis and highlight some observations relating to the query and update patterns that can be seen in this data. We stress that our aim is not a direct comparison of the performance of the agent programs or platforms analysed. The performance results presented below depend on the technology used for knowledge representation and reasoning as well as on the machine architecture used to obtain the results. As such the figures provide some insight into how these technologies perform in the context of agent programming but cannot be used directly to compare different technologies. Rather our main focus concerns the patterns that can be observed in the queries and updates that are performed by all programs and platforms, and the potential performance improvement that might be gained by caching queries on each agent platform.

We focus on the update cycles that are executed during a run of an agent. Recall that these cycles may differ from the deliberation cycle of an agent. An update cycle consists of a phase in which queries are performed which is followed by a subsequent phase in which updates are performed on the databases that an agent maintains. Note that update cycles do not correspond one-to-one to deliberation cycles. In particular, both Jason and 2APL agents execute significantly more deliberation cycles than update cycles as can be seen by comparing Table 1 with the tables below. The phases are extracted from log files by grouping query and add/del lines.

We analysed the log files to derive values for all the parameters in the abstract model introduced in Section 2, including the average number queries performed at each update cycle $N$, the average number of unique queries performed in an update cycle $K$, the average number of times that the same query is performed in an update cycle $N/K$, the average cost of a query $c_{\text{qry}}$, the average number of updates performed in an
update cycle $U$, and the average cost of an update $c_{\text{upd}}$. We also report the number of update cycles for each scenario we have run. Finally, we report the average percentage of queries that are repeated in consecutive update cycles, $p$. That is, $p$ represents the average percentage of queries that were performed in one cycle and repeated in the next update cycle.

The Jason and 2APL agents were run on a 2 GHz Intel Core Duo, 2 GB 667 MHz DDR2 SDRAM running OSX 10.6 and Java 1.6. The GOAL agents were run on a 2.66 GHz Intel Core i7, 4GB 1067 MHz DDR3, running OSX 10.6 and Java 1.6. Query and update costs are given in microseconds. The Size column in Tables 2a – 2c refers to the number of blocks in Blocks world. In Tables 3a and 3b for the Elevator Sim, Size refers to the number of people that randomly are generated by the simulator. The size column in Table 6 refers to the size of grid used: KT2 is a $6 \times 5$ grid with one pit, KT4 a $9 \times 7$ grid with 2 pits, and KT5 a $4 \times 4$ grid with 3 pits.

The results for the Blocks World environment are given in Tables 2a – 2c. Note that the average query and update costs for the GOAL agent decrease when the number of blocks increases. This effect can be explained by the fact that in this toy domain the overhead of translating queries by means of the JPL interface to SWI-Prolog queries is relatively larger in smaller sized instances than in larger sized ones. Also note that the costs found for GOAL agents cannot be used to draw conclusions about the performance of SWI-Prolog because of the significant overhead the Java interface JPL introduces.

<table>
<thead>
<tr>
<th>Size</th>
<th>N</th>
<th>K</th>
<th>n</th>
<th>$c_{\text{qry}}$</th>
<th>$c_{\text{upd}}$</th>
<th>Update cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.6</td>
<td>3.3</td>
<td>1.39</td>
<td>70%</td>
<td>376</td>
<td>16</td>
</tr>
<tr>
<td>50</td>
<td>4.8</td>
<td>3.3</td>
<td>1.46</td>
<td>82%</td>
<td>1057</td>
<td>79</td>
</tr>
<tr>
<td>100</td>
<td>5.1</td>
<td>3.3</td>
<td>1.54</td>
<td>82%</td>
<td>2788</td>
<td>152</td>
</tr>
</tbody>
</table>

(a) Jason

<table>
<thead>
<tr>
<th>Size</th>
<th>N</th>
<th>K</th>
<th>n</th>
<th>$c_{\text{qry}}$</th>
<th>$c_{\text{upd}}$</th>
<th>Update cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>41.1</td>
<td>26.8</td>
<td>1.56</td>
<td>58%</td>
<td>294</td>
<td>46</td>
</tr>
<tr>
<td>50</td>
<td>104.8</td>
<td>78.2</td>
<td>1.34</td>
<td>59%</td>
<td>273</td>
<td>230</td>
</tr>
<tr>
<td>100</td>
<td>235.4</td>
<td>165.8</td>
<td>1.42</td>
<td>59%</td>
<td>435</td>
<td>460</td>
</tr>
</tbody>
</table>

(b) 2APL

<table>
<thead>
<tr>
<th>Size</th>
<th>N</th>
<th>K</th>
<th>n</th>
<th>$c_{\text{qry}}$</th>
<th>$c_{\text{upd}}$</th>
<th>Update cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>26.0</td>
<td>17.6</td>
<td>1.48</td>
<td>59%</td>
<td>58</td>
<td>16</td>
</tr>
<tr>
<td>50</td>
<td>100.3</td>
<td>66.0</td>
<td>1.52</td>
<td>63%</td>
<td>35</td>
<td>70</td>
</tr>
<tr>
<td>100</td>
<td>153.3</td>
<td>105.7</td>
<td>1.45</td>
<td>70%</td>
<td>29</td>
<td>144</td>
</tr>
</tbody>
</table>

(c) GOAL

Table 2: Blocks World

The results for the Elevator Sim environment are given in Tables 3a and 3b. Tables 4 and 5 give the results for the MAPC 2006 & 2011 environments, and the results for the Wumpus World environment are shown in Table 6.
As can be seen, even in simple environments like the Blocks World, agent programs may perform many queries in a single update cycle (see Table 2). In the Blocks World experiments, the total number of queries performed during a run ranges from 417 queries for the GOAL agent in the small 10 blocks problem instance in only 16 deliberation cycles to 108,300 queries for the 2APL agent in the 100 blocks problem instance in 1590 deliberation cycles. Given that the Blocks World environment involves only a small number of beliefs and that the agents use only a few logical rules, this implies that the same query is repeated many times. A similar pattern can be seen in the other experiments. In all cases, the average number of times a query is performed in a single cycle is consistently larger than 1, with \( N/K \) ranging from 1.16 (Table 3b) up to 38.63 (3a). Our first observation is therefore that queries are consistently repeated in a single update cycle by all agents in all environments and across all the platforms investigated.

**Observation 1** In a single update cycle, the same query is performed more than once, i.e., we have \( n > 1 \).
A second observation that follows consistently from the data is that large percentages of queries are repeated each update cycle. We have found that 22% up to even 92% of queries are repeated in consecutive update cycles.

**Observation 2** A significant number of queries are repeated at subsequent update cycles, i.e., \( p > 20\% \).

Secondly, in all agent/environment/platform combinations investigated, in a single deliberation cycle an agent performs only a few (perhaps only one) actions that directly or indirectly change the state of the agent. This is also supported by the fact that the number of deliberation cycles in most cases is larger than the number of update cycles. In other words, the execution of a deliberation cycle does not always result in an update. Comparing the average number of updates with the average number of unique queries, we consistently find that many more queries are performed than updates in each cycle.

**Observation 3** The number of updates \( U \) (add, deletes) performed in an update cycle is significantly smaller than the number of unique queries \( K \) performed in that cycle, i.e. \( K \gg U \).

Note that all three observations are independent of the size or complexity of the environment, the complexity of the agent program or the agent platform used. This strongly suggest that the query and update performance of agent programs in the platforms investigated can be significantly improved.

## 4 Query Caching

The observations in the previous section suggest that efficiency can be significantly increased by memoization, i.e. by caching query results. The cache stores answers to queries, so that if the same query is performed again in the same update cycle, the answers can be returned without recourse to the underlying knowledge representation.

In this section, we first show how to modify the interface to the underlying knowledge representation to incorporate caching. We then extend the abstract performance model introduced in Section 2 in order to analyse the potential increase in performance of caching, and derive a relationship between \( n = N/K \) and the costs of maintaining the cache which characterises when caching is likely to be beneficial.

### 4.1 Extending the Knowledge Representation Interface

The most straightforward approach to exploit the fact that some queries are performed multiple times in a single update cycle, is to add the results of a query to the cache the first time it is performed in an update cycle, and then retrieve the results from the cache if the query is reevaluated at the same update cycle. Although very simple in requiring no information about the average number of times each unique query is repeated in a cycle, as we show below, if the cost of cache insertion is sufficiently low, significant performance improvements can be achieved. Moreover, such an approach requires only a very *loose coupling* between the cache and the underlying knowledge
representation. The cache simply acts as a filter: if a query is a cache hit the results are immediately returned by the cache; if a query is a cache miss, the query is delegated to the knowledge representation and the results stored in the cache, before being returned to the agent program.

The use of a cache requires an extension of the KRT interface with a cache operation lookup to lookup entries, an operation put to put entries into the cache, and an operation clear to clear the cache again. The basic approach can be implemented as shown in the algorithm below.

**Listing 1.1: Query Cache**

```
1  % Query Phase
2  clear(cache)
3  FOR EACH query Q; DO
4      IF lookup(Qi, answer, cache) THEN return(answer)
5      ELSE DO
6          answer = query(Qi, beliefbase)
7          put(Qi:answer, cache)
8          return(answer)
9      ENDDO
10  ENDDO
```

Of course, by only storing the query results, it is not possible to detect when cache entries are invalidated, so the cache needs to be cleared at the start of each query phase in an update cycle and rebuilt from scratch. In addition, when compiling an agent program, care is required to ensure that differences in variable names are resolved so that similar queries are retrieved from the cache instead of being recomputed. For example, the queries \( q(X,Y) \) and \( q(A,B) \) which represent the same query but use different variables should not result in a cache miss.

The cache can be implemented by a hash table. Given Observation 2, the size of the hash table can be tuned to optimal size after one or two cycles. By implementing the cache as a hash table, the insertion costs \( c_{ins} \) of an entry are constant and the evaluation costs of performing a query a second time are equal to the lookup costs, i.e. a constant \( c_{hit} \) that represents the cost for a cache hit. This results in the following performance model, adapted from the model in Section 2:

\[
K \cdot (c_{qry} + c_{ins}) + N \cdot c_{hit} + U \cdot c_{upd}
\]

It follows that whenever

\[
c_{qry} > \frac{K}{N-K} \cdot c_{ins} + \frac{N}{N-K} \cdot c_{hit}
\]

it is beneficial to implement a cache. That is, the cache increases performance whenever the average query cost is greater than the average lookup cost of a query plus the average insertion cost times the proportion of unique to non-unique queries (for \( N > K \)). As expected, the larger the average number of times \( n \) a query is performed in a single
cycle, the larger the expected efficiency gains. In the worst case in which all queries are only performed once in a cycle, i.e. \( n = 1 \), the cache will incur an increase in the cost which is linear in the number of queries, i.e. \( N \cdot (c_{ins} + c_{hit}) \).

4.2 Experimental Evaluation

To estimate values for \( c_{ins} \) and \( c_{hit} \) and the potential improvement in performance that may be obtained from caching, we implemented the caching mechanism described in algorithm 1.1 and evaluated its performance by simulating the execution of each agent platform with caching using the query and update logs for the Blocks World and Elevator Sim experiments. The cache is implemented as a single hash table that is filled the first time a query is performed in a query phase and cleared when the first update is performed in the subsequent update phase.

As might be expected, the cost of both cache insertions and hits were low. For our implementation, the cost \( c_{hit} \) was about 1 microsecond \((0.45 - 1.16 \mu s)\) with a similar value for \( c_{ins} \) \((0.29 - 0.83 \mu s)\).

Even in the experiment with the lowest value of \( n \) (the Elevator Sim agent programmed in GOAL with 10 people to be transported) the condition of equation 3 is satisfied and performance is improved by caching. In this case, \( n = 1.16 \) and \( c_Q = 29 \) (see Table 3b), and we have \( 29 > 1/0.16 + 1 = 7.25 \). The average estimated gain per cycle in this case is \( 42 \mu s \), which can be computed using equation 2 and subtracting the first from the last. The performance gained even in this case is about 10\%. In all other cases the gains of using single cycle caching are substantially larger.

5 Related Work

There is almost no work that directly relates to our study of the performance of knowledge representation and reasoning capabilities incorporated into agent programming. As far as we know, our study is the first to investigate patterns in the queries and updates that are performed by agent programs. In [15] it was observed that agent programs appear to spend most of their time in evaluating conditions for adopting plans, although the author’s proposed solution was to adopt a plan indexing scheme, rather than to optimize query evaluation in general. In [16] the performance of the FLUX and GOLOG agent programming languages is studied. Another GOLOG-style language, Indi-GOLOG, implements caching [17]. GOLOG-like languages, however, do not implement a deliberation cycle based on the BDI paradigm.

Performance issues of BDI agents have been studied in various other contexts. To mention just a few examples, [18] proposes an extended deliberation cycle for BDI agents that takes advantage of environmental events and [19] proposes the incorporation of learning techniques into BDI agents to improve their performance in dynamic environments. The focus of these papers is on integrating additional techniques into an agent’s architecture to improve the performance of an agent instead of on the KRT capabilities of those agents.
6 Conclusion

We presented an abstract performance model of the basic query and update operations that define the interface to a logic-based BDI agent’s underlying knowledge representation. Using this model, we analysed the performance of a variety of different agent programs implemented using three different agent platforms. To the best of our knowledge, our study is the first to analyse query and update patterns in existing agent programming languages. Although preliminary, our results suggest that in logic-based agent platforms, knowledge representation and reasoning capabilities account for a large part of the execution time of an agent. In particular, three key observations suggest that integrating memoization into agent programming languages have the potential to significantly increase the performance of logic-based agent platforms: the same queries are performed more than once in a single update cycle, large number of queries are repeated in subsequent cycles, and the number of queries is typically much larger than the number of updates performed.

We showed how the interface to the underlying knowledge representation of an agent platform can be modified to incorporate caching, and extended the abstract performance model to quantify the potential performance improvements that can be achieved by memoization of queries. Our results indicate that even simple query caching techniques have the potential to substantially improve the performance across a wide range of application domains.

The work presented here is limited to a single agent update cycle. Our results, and in particular the observation that a significant number of queries are repeated in subsequent agent cycles, suggests that further performance improvements may be obtained by extending caching to multiple cycles. Extending our abstract performance model and implementation to account for such queries is an area of further work.

References

Typing Multi-Agent Programs in simpAL

Alessandro Ricci and Andrea Santi

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 simple 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 [9, 5, 3]. 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 [17] 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 [14]. To authors’ knowledge, there are no agent programming languages 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 which introduces typing in agent oriented programming, in particular by means of a novel simple agent programming language called simpAL which has been conceived from scratch to have this feature. simpAL, whose general design and concepts have been already introduced elsewhere very recently [15], has been conceived on the one side drawing inspiration from existing APLs based on the BDI model [12] – AgentSpeak(L) [11] / Jason [1] in particular – and existing meta-models such as the A&A [10] (Agents and Artifacts), along with related frameworks such as
CArtAgO [13]. 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 [14]. Generally speaking, it 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 we explain what kind of errors we aim at detecting by introducing typing, as an improvement of the current error-checking support provided by existing APLs in the state of the art. Then, in Section 3 we describe the basic elements that characterize typing in simpAL. Finally in Section 4 we provide concluding remarks.

2 Bringing Types in Agent-Oriented Programming: Desiderata

What kind of static error checking mechanisms does the current state-of-the-art APLs provide? Summarizing, beside mere syntactical controls, there are APLs – e.g. Jason [1] – that do not provide any particular kind of checks, while others – such as 2APL [6], GOAL [8] and AFAPL [16] – provide some basic mechanisms for static errors detection. For example in 2APL proper 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 [2] (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.

However, the current set of basic checks available in APLs, when present, is not comprehensive. Therefore MAS developers are forced to deal at run-time 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.

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
where, due to a typo, we try to retrieve the belief iterations(\(N\)) using the predicate \(\text{num\_iterations}(N)\) – 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 run-time, where wrong means e.g. goals that are unknown by the agents. For example, let’s consider the case of agent ag2 requesting to agent ag3 the achievement of the goal floor\_cleaned (line 5 in Fig. 1 on 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 on 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 cannot 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 of mainstream languages, such as context-assist, code completion, etc.¹ — is available for download as an open-source project², 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 for this paper will be considered—the interested reader can refer to [15] and to the technical documentation on the web site for a more extensive account.

¹ Some snapshots of the IDE at work are available on simpAL web site and directly reachable at http://tinyurl.com/832o8hk
² http://simpal.sourceforge.net
3.1 simpAL Overview

The main inspiration for simpAL abstractions comes – on the one side – from the A&A model [10] and from the BDI (Belief-Desire-Intention) model, in particular from its implementation in existing APL, Jason in particular. On the other side, differently from existing BDI-based APL, 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 actually can eventually 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 [15], 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. 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.

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 [12]: “[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”([12], p. 313).
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 its 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 object. 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 with respect to 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 the possible types of tasks that any agent playing that role is able to do. Fig. 2 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 – which defines a set of tasks – is identified by a unique identifier (name) inside the role and its definition includes a set of typed attributes (e.g. targetTemp), which are used to fully describe the task. A task type instance is like a record with the attributes 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, plans, etc, which are typed too.

This 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 that role, checking that – for instance – they would request the accomplishment only of those tasks that are specified by the role.

In simpAL, the former case concerns checking 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). As an example, Fig. 3 shows an ACMEThermostat script implementing the Thermostat role. The error checking rule states informally:

– for an agent script, for each type of task $T$ defined in any role $R$ implemented by a script, it must exist one plan $P$ for $T$.

Given this rule, the ACMEThermostat script described in Fig. 3 is correct, while a script like the following one:

```java
agent-script UncompleteThermostatImpl implements Thermostat {
    plan-for AchieveTemperature { }
    plan-for DoSelfTest { }
}
```
agent-script ACMEThermostat implements Thermostat in SmartHome {
    cond: Conditioner;
    therm: Thermometer;
    threshold: double;
    condSpeed: double;

    plan-for Init(conditioner: Conditioner thermometer: Thermometer) {
        cond = conditioner;
        therm = thermometer;
        threshold = 1;
        condSpeed = 0.50;
    }

    plan-for AchieveTemperature using: cond, therm, log @ mainRoom {
        log(msg: "achieving temperature \"targetTemp\" from \"currentTemp\")
        completed-when: java.lang.Math.abs(targetTemp - currentTemp) < threshold {
            every-time currentTemp > (targetTemp + threshold) && !isCooling => startCooling(speed: condSpeed)
            every-time currentTemp < (targetTemp - threshold) && !isHeating => startHeating(speed: condSpeed)
        }
        if (isHeating || isCooling) {
            stop()
        }
    }

    plan-for KeepTemperature using: cond, therm, userView {
        quitPlan : boolean = false;
        completed-when: quitPlan {
            do-task AchieveTemperature(targetTemp: desiredTemp)
            every-time changed desiredTemp => atomically: {
                if (is-doing-any AchieveTemperature) {
                    drop-all-tasks AchieveTemperature
                }
                do-task AchieveTemperature(targetTemp: desiredTemp)
            }
        }
        every-time changed currentTemp && is-doing-any AchieveTemperature => do-task AchieveTemperature(targetTemp: desiredTemp)
        every-time told newThreshold => {
            threshold = newThreshold
        }
        every-time changed thermostatStatus : thermStatus.equals == ThermostatStatus.OFF => atomically: {
            if (isCooling || isHeating) {
                stop()
            }
            drop-all-tasks AchieveTemperature
            quitPlan = true
        }
    }

    plan-for SelfTest() using: log @ mainRoom, cond {
        log(msg:"Simple thermostat \"myName\")
        log(msg:"Using the conditioner \" + name in cond)
    }
}

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 \texttt{ACMThermostat} 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 \texttt{ACMThermostat} script has four beliefs, two to keep track of the conditioner and thermometer artifacts used to do his job, one for the threshold and one for the speed to be used when using the conditioner. Beliefs declared at the script level are a sort of long-term memory of the agent, useful to keep track 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 \texttt{ACMThermostat} script has four plans, to achieve a certain temperature, to maintain a temperature, to do some self test, and to initialize the script, which is executed by default when the script is loaded.

To do the \texttt{AchieveTemperature} task, the plan starts cooling or heating – using the conditioner – as soon as the current temperature is high or low compared to the threshold. The belief about the target temperature (\texttt{targetTemp}) derives from the related attribute (parameter) of the task, while the one about the current temperature (\texttt{currentTemp}) is related to the observable property of the \texttt{therm} thermometer 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 \texttt{KeepTemperature} task, the plan achieves the desired temperature by immediately executing the sub-task \texttt{AchieveTemperature} (line 29), which is executed also as soon as the desired temperature changes (lines 31-36) or the current temperature changes and the agent is not already achieving the temperature (line 38-39). The belief about the desired temperature (\texttt{desiredTemp}) comes from the observable property of the \texttt{userView} artifact used in the plan. Also, as soon as a message about a new threshold is told by some other agent, the internal value of the threshold is updated. The plan quits if the agent perceives from the \texttt{userView} artifact that the user has switched off the thermostat.

In that case, before quitting the plan, the conditioner is eventually stopped if it was working. Finally, to do the \texttt{SelfTest} task, the agent simply logs on the console a sequence of information, about his identifier, and the name of the conditioner which is using.

Some explanations about some key elements of the syntax and semantics of plans follow — a more comprehensive description can be found here [15] 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 events that the previous one has completed. In the most general case, an action rule is of the kind: 
\begin{quote}
\texttt{ every-time} | \texttt{when Event} : \texttt{Condition} \rightarrow \texttt{Action} meaning that the specified action can be executed every time or once that (when) the specified event occur and the specified condition – which is a boolean expression of the agent beliefs base – holds. If not specified, the default value of the condition is true. Events concern percepts related to either one of (a) the environment, (b) messages sent by agents, (c) action execution, (d) time passing. All events are actually uniformly modeled as changes to some belief belonging to agent belief base, given the fact that observable properties, messages sent, action state variables, the internal clock of the agent are all represented as beliefs. Furthermore, the syntax for specifying events related to a change of an observable property is \texttt{changed \texttt{ObsProp}} (e.g. line 38), the one for specifying the update of a belief about an information told by another agent is \texttt{told \texttt{What}} (e.g. line 42). If no event is specified, the pre-defined 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. Actions can be either external actions to affect the environment, i.e. operations provided of some artifact, or communicative actions to directly interact with some other agent (to tell some belief, to assign a task, etc.) or predefined internal actions – to update internal beliefs, to manage tasks in execution, etc. An action can be also an action rule block \{\ldots\}, 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 \texttt{using}: attribute to specify the list of the identifiers of the artifacts used inside the block (an artifact can be used/observed only if explicitly declared), the \texttt{completed-when}: to specify the condition for which the action rule block execution can be considered completed, the \texttt{atomically}: 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). \end{quote}
would report an error message about missing a plan for a declared task, i.e. KeepTemperature.

The second case concerns checking tasks assigned to an agent of type $R$, i.e. which is expected to play the role $R$. Task assignment can be done in two ways. The first is through a predefined communicative action do-task done by any agent specifying the typed identifier of the task recipient:

$$\text{do-task TaskTodo task-recipient: Id}$$

The second is through a pre-defined internal action with the same name, without specifying the target agent, so as for an agent to allocate the task to himself. In both cases, the action completes as soon as the task has been done or it has failed\(^4\). Given this, we can enforce that:

- given a belief $Id$ of type $R$, storing the identifier of some agent playing the role $R$, then for any communication action do-task $t$ task-recipient: $Id$ that sends a message to the target agent to assign him the task $t$, there must exist a task type $T$ in $R$ such that $t$ is a value (instance) of $T$.

Then, given a fragment of a script with e.g. a belief myThermostat declared of type Thermostat, we have the following list of the main kind of errors that can be caught at compile time:

```plaintext
/* compilation ok */
do-task AchieveTemperature( targetTemp: 21 ) task-recipient: myThermostat

/* error: no tasks matching CleanTheRoom in role Thermostat */
do-task CleanTheRoom() task-recipient: myThermostat

/* error: no targetT param in AchieveTemperature */
do-task AchieveTemperature( targetT: 21 ) task-recipient: myThermostat

/* error: wrong type for the param value targetTemp */
do-task AchieveTemperature( targetTemp: "21" ) task-recipient: myThermostat
```

The definition of role types includes also the type of messages that can be sent to an agent playing the role, in particular what kind of beliefs can be told by other agents. It can be declared by understands blocks. In the example, agents playing the Thermostat role can be told about the new threshold to adopt, which is represented by a belief newThreshold containing a value of type double (see lines 13-15 in Fig. 2). This declaration allows for checking at compile time that the communicative actions of agents aiming at informing other agents about some belief. In simpAL this can be done by the communicative action:

```
tell Bel = BelExpr to: Id
```

\(^4\) in the case of the task assigned to another agent this is done by a message which is automatically sent from the recipient to the task allocator.
that sends a message to the receiver agent \(Id\) of some role type \(R\) to tell him that the value of the belief \(Bel\) is the value of the expression \(BelExpr\). Given that, we can check then that the specified \(Bel\) would be among those that are enlisted in the \textit{understands} block of the role \(R\) and that the types of the beliefs are compatible. Examples of checks follow:

```/* compilation ok */
tell newThreshold = 2 to: myThermostat
```

```/* error: aMsg is not understood by agent playing the Thermostat role */
tell aMsg = "hello" to: myThermostat
```

```/* error: wrong type for the belief newThreshold told to a Thermostat*/
tell newThreshold = "2" to: myThermostat
```

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 that beliefs are assigned with expressions of wrong type.

### 3.3 Typing the Environment

On the environment side, we introduce the notion of artifact \textit{interface} defining the type of the artifacts, separated from its implementation provided by artifact \textit{templates}. An artifact 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 by \texttt{Thermostat} agents, namely \texttt{Conditioner} – representing the interface of conditioner devices modeled as artifacts, used by thermostat agents to heat or cool – \texttt{Thermometer} – used by agents to be aware of the current temperature – and \texttt{UserView} – representing the interface of those GUI artifacts used to interact with the human users, in particular to know what is the desired temperature.

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), checking that it conforms to its specification (the interface).

In \texttt{simpAL}, the former case concerns checking the action (rules) in plan bodies, so that for each action \texttt{OpName(Params) on Target}, specified in an action rule, meaning the execution of an operation \texttt{OpName} over an artifact identifier \texttt{Target} whose type is \(I\):

- there must exist an operation defined in the interface \(I\) matching the operation request;
artifact ACMEConditioner implements Conditioner {
    nTimesUsed: int;

    init(){
        isCooling = false; isHeating = true; nTimesUsed = 0;
    }

    startCooling(speed: double){
        nTimesUsed++;
        isCooling = true; isHeating = false;
    }

    startHeating(speed: double){...}

    stop(){
        isCooling = false; isHeating = false;
    }
}

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.

– 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:

/* compilation ok */
startCooling (speed: 0.75) on cond

/* error: unknown operation switchOn */
switchOn () on cond

/* error: unknown parameter time: in startCooling operation */
startCooling (speed: 0.75 time: 10) on cond
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 ACMThermostat 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 and its type.

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:

/* compilation ok */
updated currentTemp => println(msg: "the temperature has changed")
updated currentTemp : isHeating => println(msg: "the temperature has changed while heating...")
sum: double = currentTemp in therm + 1

/* error: unknown obs property isHeating in Thermometer type */
updated isHeating in therm => ...

/* error: wrong type */
bak: boolean = currentTemp in therm

On the environment side 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 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 to define the structure of the overall multi-agent program, and we introduce then the type of an organization called \textit{organization model} to explicitly declare the set of roles and interfaces which are used inside any organization of that type, and the set of workspaces that are available in any organization of that type. The declaration of the workspace can include then explicit declaration of the identifier (literals) of instances of artifacts and agents – along with their types – that are known to be available in that workspace\(^5\). Such identifiers are like global references that can be then referred in any agent script (so at 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 shows the definition of the \textit{SmartHome}, with a \texttt{mainRoom} workspace hosting an artifact \texttt{log} of type \texttt{Log} and an agent \texttt{majordomo} of type \texttt{HomeAdmin}. Given this org model definition, then it is possible e.g. in the \texttt{KeepTemperature} plan of the \texttt{ACMThermostat} script to refer directly to the artifact \texttt{log} \texttt{@ mainRoom} (in spite of its actual implementation).

\(^5\) In general, a workspace can contain at runtime also agents/artifacts not declared in the model: their types, however, must be among the types described in the model.
Given an organization model, it is possible then to define concrete organizations that implements the model and eventually specify the initial configuration by instantiating the components (artifacts, agents) specified in it\(^6\). 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.

As a final remark, the notion of organization used here is not meant to be as rich as to one which appears in MAS organization modelling; the main objective of introducing this concept here is to have a way to define rigorously the structure 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 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 where 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 allows a first level of reuse and evolvability, without the need of having the source codes. Improved version of the implementation 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 [17]. 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 [4] – 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 [7].

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

\(^6\) Not all the agents and artifacts of a program needs to be specified and created in the main organization file, other can be eventually spawned and created dynamically.
for eventually refining and enriching how such concepts are currently modeled in simpAL.

References

Learning to Improve Agent Behaviours in GOAL

Dhirendra Singh and Koen V. Hindriks
Interactive Intelligence Group, Delft University of Technology, The Netherlands

Abstract. This paper investigates the issue of adaptability of behaviour in the context of agent-oriented programming. We focus on improving action selection in rule-based agent programming languages using a reinforcement learning mechanism under the hood. The novelty is that learning utilises the existing mental state representation of the agent, which means that (i) the programming model is unchanged and using learning within the program becomes straightforward, and (ii) adaptive behaviours can be combined with regular behaviours in a modular way. Overall, the key to effective programming in this setting is to balance between constraining behaviour using operational knowledge, and leaving flexibility to allow for ongoing adaptation. We illustrate this using different types of programs for solving the Blocks World problem.

Keywords: Agent programming, rule selection, reinforcement learning

1 Introduction

Belief-Desire-Intention (BDI) [1] is a practical and popular cognitive framework for implementing practical reasoning in computer programs, that has inspired many agent programming languages such as AgentSpeak(L) [2], JACK [3], Jason [4], Jadex [5], CANPLAN [6], 3APL [7], 2APL [8], and GOAL [9], to name a few. Despite its success, an important drawback of the BDI model is the lack of a learning ability, in that once deployed, BDI agents have no capacity to adapt and improve their behaviour over time. In this paper, we address this issue in the context of BDI-like rule-based agent programming languages. Particularly, we extend the GOAL agent programming language [9] for practical systems [10, 11] with a new language primitive that supports adaptive modules, i.e., modules within which action choices resulting from programmed rules are learnt over time. While we have chosen GOAL for this study, our approach applies generally to other rule-based programming languages. We use an off-the-shelf reinforcement learning [12] mechanism under the hood to implement this functionality.

Our aim is to allow agent developers to easily program adaptive behaviours using a programming model that they are already familiar with, and without having to explicitly delve into machine learning technologies. Our idea is to leverage the domain knowledge encoded into the agent program, by directly using the mental state representation of the agent for learning purposes. This has the key benefits that (i) the programmer need not worry about knowledge representation for learning as a separate issue from programming; (ii) the programming model
for adaptive behaviours remains the same as before; and (iii) learning and programming become truly integrated in the sense that their effectiveness depends directly on the mental state representation used by the programmer.

The key idea is that learning may exploit the underspecification that is inherent in agent programming [13]. That is, agent programs often generate multiple options for actions without specifying how to make a final choice between these options. This is a feature of agent programming because it does not require a programmer to specify action selection to unnatural levels of detail. The motivation of our work is to exploit this underspecification and potentially optimize action selection by means of automated learning where it may be too complicated for a programmer to optimize code. The first challenge is to add a learning mechanism to agent programming in a generic and flexible way and to naturally integrate such a mechanism in a way that burdens the programmer minimally. The second challenge is to do this in such a way that the state space to be explored by the learning mechanism can still be managed by the program. Our approach addresses both challenges by re-using the mental state representation available in the agent program. Although our approach also facilitates managing the state space, there remain issues for future work that need to be dealt with in this area in particular. To this end, we draw some lessons learned from our work and discuss some options for dealing with this issue.

One of the aims of our work is to explore the impact of various representations or program choices on the learning mechanism. Even though our objective is to impose minimal requirements on the programmer’s knowledge of machine learning, the program structure will have impact on the learning performance. Ideally, therefore, we can give the programmer some guidelines on how to write agent programs that are able to effectively learn. It is well-known that the representation language is a crucial parameter in machine learning. Given an adequate language, learning will be effective, and given an inadequate one learning will be difficult if not impossible [14]. Applied to agent-oriented programming this means that it is important to specify the right predicates for coding the agent’s mental state and to provide the right (modular) program structure to enhance the effectiveness of learning. If the programmer is not able to use knowledge to guide program design, one may have to search a larger space, may require more examples and time, and in the worst case, learning might be unsuccessful.

The remainder of the paper is as follows. Section 2 introduces the GOAL language and reinforcement learning, followed by an overview of related works in Section 3. Section 4 describes the integration of GOAL and reinforcement learning and Section 5 presents experiments in the Blocks World. We conclude with a discussion of limitations and future directions of this work in Section 6.

2 Preliminaries

We now briefly discuss how agent programs and cognitive architectures select the action to perform next. In other words, we discuss how the mechanism we want to extend with a learning capability works. Following this, we introduce the reinforcement learning framework that we have used in this work.
2.1 Agent Programming Languages

Agent programming languages (APLs) based on the BDI paradigm are rule-based languages [15, 16]. Rules may serve various purposes but are used among others to select the actions or plans that an agent will perform. An agent program may perform built-in actions that are provided as programming constructs that are part of the language itself or it may perform actions that are available in an environment that the agent is connected to. Environment actions give the agent some control over the changes that occur in that environment. The types of rules that are used in APLs varies. Generally speaking, however, rules have a condition that is evaluated on the agent’s mental state (rule head) and have a corresponding action or plan that is instantiated if the rule fired (rule body).

Rules are evaluated and applied in a reasoning cycle that is part of the agent program’s interpreter. Agent interpreters for APLs implement a sense-plan-act cycle or a variant thereof. In a typical interpreter, for example, the percepts received from an environment are processed by some predefined mechanism. Most often this is an automatic mechanism (that may be customisable as in Jason) [4], but not necessarily; in GOAL, for example, so-called percept rules are available for processing incoming percepts and similar rules are available for processing messages received from other agents (2APL has similar rules [8]). During this stage, either before or after processing the percepts, typically the messages received from other agents are processed. These steps are usually performed first to ensure the mental state of the agent is up to date. Thereafter, the interpreter will evaluate rules against the updated mental state and select applicable rules. After determining which rules are applicable one or more of these rules is fired, resulting in one or more options to perform an action or add a plan to a plan base. Some selection mechanism (that again may be customised as e.g. in Jason) then is used to arbitrate between these multiple options, or, as is the case in for example GOAL, a choice is randomly made. Finally, an action is executed either internally or an action is sent to the environment for execution.

One aspect of these interpreters is that they may generate multiple applicable rules and options for performing actions. If multiple options are available, then the agent program is underspecified in the sense that it does not determine a unique choice of action. It is this feature of agent architectures that we will exploit and can be used by a learning mechanism for optimising the agent’s choice of action [17, 18].

2.2 Reinforcement Learning

Reinforcement learning [12] is a formal framework for optimally solving multistage decision problems in environments where outcomes are only partly attributed to decision making by the agent and are partly stochastic. The general idea is to describe the value of a decision problem at a given time step in terms of the payoffs received from choices made so far, and the value of the remaining problem that results from those initial choices. Formally, at each time step $t$ in a multistep problem, the agent perceives state $s \in S$ of the environment and
chooses an action $a \in A_s$ that causes the environment to transition to state $s'$ in the next time step $t+1$ and return a reward with the expected value $r \in R(s, a)$. Here $S$ is the set of all possible states, $A_s$ is the set of all possible actions in a given state $s$, and $R(s, a)$ is the function that determines the reward for taking an action $a$ in state $s$. The probability that the process advances to state $s'$ is given by the state transition function $P(s'|s, a)$. The agent’s behaviour is described by a policy $\pi$ that determines how the agent chooses an action in any given state. The optimal value in this setup can be obtained using dynamic programming and is given by Bellman’s equation (Equation 1) [19], that relates the value function $V^*(s)$ in one time step to the value function $V^*(s')$ in the next time step.

$$V^*(s) = R(s, a) + \max_{a \in A_s} \gamma \sum_{s'} P(s'|s, a) V^*(s').$$

(1)

In the reinforcement learning setting both $P(s'|s, a)$ and $R(s, a)$ are unknown. So the agent has little choice but to physically act in the environment to observe the immediate reward, and use the samples over time to build estimates of the expected return in each state, in the hope of obtaining a good approximation of the optimal policy. Typically, the agent tries to maximise some cumulative function of the immediate rewards, such as the expected discounted return $R^\pi(s)$ (Equation 2) at each time step $t$. $R^\pi(s)$ captures the infinite-horizon discounted (by $\gamma$) sum of the rewards that the agent may expect (denoted by $E$) to receive starting in state $s$ and following the policy $\pi$.

$$R^\pi(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots\}. \quad (2)$$

One way to maximise this function is to evaluate all policies by simply following each one, sampling the rewards obtained, and then choosing the policy that gave the best return. The obvious problem with such a brute force method is that the number of possible policies is often too large to be practical. Furthermore, if rewards were stochastic, then even more samples will be required in order to estimate the expected return. A practical solution, based on Bellman’s work on value iteration, is Watkins’ Q-Learning algorithm [20] given by the action-value function (Equation 3). The $Q$-function gives the expected discounted return for taking action $a$ in state $s$ and following the policy $\pi$ thereafter. Here $\alpha$ is the learning rate that determines to what extent the existing $Q$-value (i.e., $Q^\pi(s, a)$) will be corrected by the new update (i.e., $R(s, a) + \gamma \max_{a'} (Q(s', a'))$), and $\max_{a'} (Q(s', a'))$ is the maximum possible reward in the following state, i.e., it is the reward for taking the optimal action thereafter.

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \alpha \left[ R(s, a) + \gamma \max_{a'} (Q(s', a')) - Q^\pi(s, a) \right]. \quad (3)$$

In order to learn the $Q$-values, the agent must try out available actions in each state and learn from these experiences over time. Given that acting and learning are interleaved and ongoing performance is important, a key challenge when
choosing actions is to find a good balance between exploiting current knowledge to get the best reward known so far, and exploring new actions in the hope of finding better rewards. A simple way to achieve this is to select the best known action most of the time, but every once in a while choose a random action with a small probability, say $\epsilon$. This strategy is well known as $\epsilon$-greedy and is the one we use in this work. In future work we plan to experiment with more advanced strategies. For instance, in the so-called Boltzmann selection strategy, instead of picking actions randomly, weights are assigned to the available actions based on their existing action-value estimates, so that actions that perform well have a higher chance of being selected in the exploration phase.

In this study we use a $Q$-Learning implementation where the precise action-value function is maintained in memory. It should be noted here that this implementation does not scale well to large state spaces. Of course, we could use an approximation of the action-value function, such as a neural network, to store this in a compact manner. However, our focus here is not so much to use an efficient reinforcement learning technology as it is to see how such learning can be integrated into agent programming in a seamless manner. For this reason, in this version of the work, we have kept the basic $Q$-Learning implementation.

3 Related Work

In most languages for partial reinforcement learning programs, the programmer specifies a program containing choice points [21]. Because of the underspecification present in agent programming languages, there is no need to add such choice points as multiple options are generated automatically by the agent program itself. There is little existing work in integrating learning capabilities within agent programming languages. In PRS-like cognitive architectures [2, 4, 22, 3] that are based in the BDI tradition, standard operating knowledge is programmed as abstract recipes or plans, often in a hierarchical manner. Plans whose preconditions hold in any runtime situation are considered applicable in that situation and may be chosen for execution. While such frameworks do not typically support learning, there has been recent work in this area. For instance, in [23] the learning process that decides when and how learning should proceed, is itself described within plans that can be invoked in the usual manner. Our own previous investigations in this area include [24–26] where decision tree learning was used to improve hierarchical plan selection in the JACK [3] agent programming language. That work bears some resemblance here in that the aim was to improve choice of instantiated plans as we do for bound action options in this study. In [17] we integrated GOAL and reinforcement learning as we do in this paper, with the key difference that now (i) a learning primitive has been added to the GOAL language to explicitly support adaptive behaviours, and (ii) a much richer state representation is used, i.e., the mental state of the agent.

Among other rule-based systems, ACT-R [27, 28] is a cognitive architecture primarily concerned with modelling human behaviour, where programming consists of writing production rules [29] that are condition-action pairs to describe possible responses to various situations. Learning in ACT-R consists of forming
entirely new rules from sample solutions encountered, as well as updating the utilities of existing rules from ongoing experience. While not a programming language per se, ACT-R learning is nevertheless quite related in that Q-Learning is also used to learn rule preferences. SOAR [30] also uses production rules to capture procedural knowledge about the domain. It uses a process called chunking to create new production rules based on the results of subgoals, in a kind of explanation-based learning. SOAR-RL [31] integrates reinforcement learning to improve operator selection based on experience, similar to learnt utilities in ACT-R. ICARUS [32] is a cognitive architecture that incorporates ideas from work on production systems, hierarchical task networks, and logic programming. It uses a form of explanation-based learning to find the task hierarchy in hierarchical task networks. Overall, ACT-R, SOAR and our work share similarities in the way reinforcement learning is used to learn rule preferences, however the motivations are quite different. While ACT-R is heavily used in cognitive psychology research to model human behaviour, and SOAR is a general cognitive architecture for building intelligent systems, GOAL is an agent programming language in the BDI tradition. For us, the key motivation for integrating learning is to make adaptive technologies more accessible to agent programmers.

4 The GOAL Agent Programming Language

GOAL is a logic-based agent programming language similar to 2APL [8] and Jason [4]. GOAL agents maintain a dynamic mental state consisting of beliefs and goals that are represented in Prolog. GOAL agents also have a static knowledge base that is part of their mental state and consists of domain knowledge. They may perform built-in actions that update their mental state or send a message as well as actions that are available in the environment that the agent is connected to. Environment actions are specified using a STRIPS-like pre- and post-condition specification. A GOAL agent derives its choice of action from its beliefs and goals (in combination with its knowledge) by means of rules. Rules consist of a condition that is evaluated on the mental state of the agent and one or more actions that may be executed if the condition holds. In addition, GOAL supports multiple types of rules, rule evaluation strategies, and modules that facilitate structured programming.

Figure 1 provides a listing of a simple example GOAL agent program for the Blocks World [33]. We have used this program also in our experiments to evaluate the learning mechanism we have added to the language. The Blocks World is a well studied toy domain that has been used extensively in artificial intelligence research. The setup consists of a fixed number of blocks that are sitting on a table big enough to hold them all. Each block exists on top of exactly one other object that can either be another block or the table itself. Each block is considered to be clear when no other block exists on top of it. There is only one type of action that is possible in this domain: move a single clear block, either from another block onto the table, or from an object onto another clear block. A problem specification in this domain consists of an initial configuration of blocks, as well
as the desired configuration. The task for the agent is to move blocks around one at a time until the final configuration is realised.

The program processes percepts and randomly selects an action that is enabled in the Blocks World environment. The init module consists of code to initialize the mental state of the agent and a single action specification for the move(X,Y) action that the agent can perform in the Blocks World. The Prolog rules in the knowledge section of this module define the concepts of a block and a block being clear. An initial goal is specified in the goals section. Initially the agent has no beliefs; the agent must first perceive the environment to obtain information about the blocks’ configuration. The event module is executed at the start of each decision or reasoning cycle of an agent. Its purpose is to process received percepts (and messages). The two forall rules part of this module process the percepts received at the start of the cycle of the form percept(on(X,Y)). The first rule checks whether the agent sees that a block X is on top of a block Y and inserts this fact if the agent does not currently believe it; the second rule removes facts that are believed but not perceived (assuming full observability this is a sound rule).

The main module consists of the decision logic for acting in the environment and selects actions after the mental state has been updated with the most recent perceptual information. The option order=random associated with the program
section of the module indicates that rule evaluation occurs in random order (other options are to evaluate in linear order and to evaluate all rules in either linear or random order). In the example agent of Figure 1 there is only one rule that is always applicable because the condition \texttt{bel(true)} always holds and there is always an action enabled in a Blocks World environment. Note that a rule is evaluated by evaluating its condition \textit{and} the precondition of the (first, if there are more) action of the rule. A rule that is applicable generates a non-empty set of \textit{options} which are actions that may be performed by the agent. Only one of these actions is performed in a rule of the form \texttt{if \ldots then \ldots} and the action is randomly selected in that case. One might say that the program underspecifies what the agent should do. This may be useful if a programmer does not know how to select or care for a unique action and may be exploited by a learning mechanism to optimise the behavior of the agent, which is exactly the focus of this paper. Where more than one action is applicable, the agent must decide which one to choose and execute. In this work we are concerned with improving this action selection based on the ongoing experiences of the agent.

**Implementing Adaptive Behaviours in GOAL**

There are two key aspects of the adaptive framework in GOAL that we describe now, (i) the programming model that describes how adaptive behaviour modules can be specified by the agent programmer using the language of GOAL, and (ii) the underlying reinforcement learning implementation that makes it all possible.

Let us start with how adaptive behaviours can be specified in the programming model of GOAL. An important consideration for us when deciding what the programming interface should consist of was to keep as much of the machine learning technologies insulated from the programmer as possible. The motivation for this was to keep the programming model as simple and as close to the existing model as possible in order to allow easy uptake by existing GOAL programmers. Our stance here has been that agent programmers are not experts in machine learning and so they will be reluctant to try a new feature of the language if it required new expertise and significant changes to their programming style.

The first design choice for us was how the knowledge representation aspect of machine learning should be combined in the agent programming model without sacrificing programming flexibility and avoiding significant overhead. \textit{We achieve this by making the knowledge representation for learning to be the same as the knowledge representation for the agent program.} That is to say that the “state” in the reinforcement learning sense is the mental state of the agent that comprises of its beliefs and goals. All that is required is to provide a translation function that automatically maps the mental state of the agent to a suitable state id (a number) used for reinforcement learning. The second decision was to make learning a modular feature in line with the GOAL philosophy of modular programming, to allow regular and adaptive behaviours to be easily combined.

The result is a very easy to use programming model where learning can simply be enabled as desired using a new \texttt{order=adaptive} option in the program section.
of a module. For example, to change the regular program module in the agent of Figure 1 to an adaptive one, we only have to change \texttt{order=random} as follows:

```java
main module { program[order=adaptive] {
  if bel(true) then move(X,Y).
}}
```

With this specification, all possible action bindings will be evaluated by the underlying learning mechanism and action selection will be governed by the \(Q\)-values derived over repeated runs, rather than being random as it was before.

The benefit is that the agent programmer does not have to explicitly think about learning as being separate from programming, bar adhering to some basic guidelines. The only recommendation we have for the programmer is to not use the belief base of the agent as a long term memory store if learning is to be used. This means to not keep adding belief facts that only serve to keep a history of events. For example, a programmer may choose to store the history of every choice it made in a maze solving problem. If the programmer then enables learning such as to optimise the maze exploration strategy, then it will likely not deliver any useful results quickly due to the very large state space created by the belief base. A similar argument also applies for adding new goals to the mental state, but it is generally not as much of a problem since programs do not add new goals during execution to the same extent as they do beliefs.

We must add here that in some problems this representation is unavoidable. In future work, we hope to address such cases by allowing learning to decouple the mental state into relevant and not relevant parts using a dependency graph that is already part of the GOAL implementation. For instance, in the maze example, if the exploration module code does not depend on the history of beliefs being added, then it should be possible to automatically isolate them from the state representation for learning purposes.

The final decision was on how rewards should be specified for reinforcement learning within the GOAL programming framework. We do this using the existing Environment Interface Standard (EIS) that GOAL uses to connect to external environments. The addition is a new “reward” query from GOAL that, if implemented by the environment, returns the reward for the last executed action. If, however, the plugged environment does not support this query, then the reward is simply an evaluation of the goals of the agent: if all the goals have been achieved, then the reward is 1.0, otherwise it is 0.0. The idea is that learning can be enabled regardless of whether a reward signal is available from the environment, in which case the agent tries to optimise the number of steps it takes to achieve its goals. A future extension might be to give partial rewards between 0.0 and 1.0 based on how many independent goals have been satisfied. However, it is unclear if rewards based solely on the agent’s goals are always useful in learning, such as in programs that add or remove goals from the mental state.

Under the hood we have implemented a Java-based interface that allows us to plug in a generic reinforcement learning algorithm into GOAL. The idea is to be able to offload the task of providing and maintaining the machine learning technology to the relevant expert. It will also allow us to easily update the default \(Q\)-Learning implementation with a more efficient one in the future.
5 Experiments

Here we describe the Blocks World domain that we used as a testbed for our experiments, and then the three different programs to solve it. We analyse the results quantitatively in terms of the average number of steps taken by the agent to achieve its goal, as well as qualitatively in terms of how the design of the program impacts learning performance.

We have chosen the Blocks World domain for our experiments for several reasons. First, the domain is simple to understand and programming strategies are easy to describe and compare at a conceptual level. Second, despite its simplicity, finding optimal solutions in this domain is known to be an NP-hard problem [34]. Finally, decisions in this domain often involve choosing between several options that could potentially be optimised using learning.

There are various ways of programming a strategy for solving the Blocks World. For example, one way would be to dismantle all blocks onto the table one by one, and then stack them into the desired configuration from there. This is in fact a reasonable “baseline” strategy because it is easy to see that the upper bound for the number of steps needed to solve a problem with \( n \) blocks is \( 2(n - 1) \) which is the case when one must dismantle a single tower (which takes \( n - 1 \) moves for a tower of height \( n \)) to construct a different single tower (that takes another \( n - 1 \) moves). The average number of steps for this algorithm is less intuitive but has been shown to be \( 2(n - \sqrt{n}) \) [33]. For this work, we will compare three other solutions to the problem, and see how they compare amongst themselves and against this baseline strategy.

**Program A** A very simple strategy for solving the Blocks World is to randomly select some block that is clear and move it to some randomly chosen place on top of another object. Effectively, this strategy tries to achieve the final configuration by randomly changing the current configuration for as long as needed until it eventually stumbles upon the solution. This strategy is given by the program listing in Figure 1, and is contained in the following code segment:

```plaintext
main module { program[order=random] {
    if bel(true) then move(X,Y).
}}
```

This is certainly not the most effective way to solve the problem, and while it works reasonably well for small problems of two to four blocks, it quickly becomes unusable beyond six blocks. Nevertheless it is useful for this study since we are interested in improving action selection using learning, and one would imagine there is a lot of room for improvement in this strategy.

**Program B** An improvement on the random strategy is this actual Blocks World program written in GOAL by an agent programmer:

```plaintext
main module { program[order=random] {
    if bel(on(X,Y), clear(X), clear(Z)), a-goal(on(X,Z)) then move(X,Z).
    if bel(on(X,Y), not(clear(X))), a-goal(on(X,Z)) then adopt(clear(X)).
    if a-goal(on(X,Z)), bel(on(X,Y), not(clear(Z))) then adopt(clear(Z)).
```
This strategy uses the following line of thought: If the agent has a goal to have some block $X$ on top of $Z$, then move $X$ onto $Z$ if possible. If not possible because $X$ cannot be moved, then clear whatever block is obstructing $X$. On the other hand, if it is $Z$ that is blocked then clear it first. Finally, repeatedly clear blocks that are obstructing other blocks that are to be cleared.

**Program C** A more sophisticated solution that comes bundled with the GOAL distribution uses a higher level notion of *misplaced blocks* to decide if a block should be moved. To do this it provides a recursive definition of a *Tower*. Then a block is considered misplaced if the agent still has a goal to have a tower with block $X$ on top. Given these definitions, the strategy is relatively simple and uses only two rules. The idea is to either move a misplaced block onto the table, or move a block onto another block if the move is constructive, i.e., results in a desired tower configuration.

```plaintext
knowledge{
  ...
  tower([X]) :- on(X, table).
  tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
}
program[order=linear] {
  #define misplaced(X) a-goal(tower([X| T])).
  #define constructiveMove(X,Y) a-goal(tower([X, Y| T])), bel(tower([Y| T])).
  if constructiveMove(X, Y) then move(X, Y).
  if misplaced(X) then move(X, table).
}
```

We conducted several experiments with the three example programs A, B, and C, for problems with up to 10 blocks. Each run of the experiment consisted of a series of randomly generated problems that were solved using the program first in its original form and then using adaptive ordering (i.e., by substituting `[order=adaptive]` in the `program` module options). Since problems are randomly generated and the number of moves required to solve them can vary significantly, we used a moving average of 20 results over the series of generated problems to get the average number of steps for *any* problem of a given size. Finally, we ran 20 repeats of each experiment and report the average number of moves taken to achieve the fixed goal of building a given tower configuration.

For all of our experiments, we used the following parameters’ settings. The $\epsilon$ value for the action selection strategy was set to always explore 10% of the time. For $Q$-Learning we set the learning rate $\alpha$ to 1.0 and the discount factor $\gamma$ to 0.9. It should be noted that these settings will obviously impact learning, and these default values may not work as well in other domains. An option in the future might be to setup learning “profiles” that the programmer can select between based on some basic usage guidelines.
Results

Figure 2 shows the results of running the programs A, B, and C, with and without adaptive behaviours enabled (in dark and light shading respectively). Figure 2a, Figure 2b, and Figure 2c are results for problems with four blocks, while Figure 2d, Figure 2e, and Figure 2f are for problems with six blocks.

Program A: In Figure 2a, the light shading shows that the average number of steps taken by the original A to solve problems with four blocks is around 350 moves. This is not very surprising since A is really only trying to solve the problem using random moves. The dark shading shows the results for the same set of problems and the same A, but using adaptive rule ordering. While initially the program performs similarly as it tries to find the first few solutions, it improves to around seven moves per problem by 100 episodes. Beyond that it improves progressively and by the end of the experiment at 2000 episodes the program takes around five moves per problem. Compared to our baseline program that averages $2(n - \sqrt{n}) = 4$, i.e., four moves for a problem with four blocks, we can already see that the learnt ordering gives competitive perfor-
mance. Figure 2d shows the same A for problems with six blocks using adaptive ordering. We have not included results for the original program since it takes over 30000 moves on average per problem. For adaptive mode, this number improves to about 60 moves by the 100th episode, and progressively to around 12 moves by 4000 episodes. This gets us close to the baseline of \(2(n - \sqrt{n}) = 7.1\) but not quite there. It would be possible to improve further if the program was allowed to run for more episodes, but the improvement will occur very slowly. We also did not run this program for problems with more than six blocks as solving larger problems becomes impractical with this strategy.

**Program B:** Figure 2b shows the performance of the original B for problems with four blocks at around 11 moves. The performance is already reasonable to start with as it is a more informed programmed strategy than A. With adaptive ordering, the performance improves to around five moves per problem by 100 episodes. This is on par with the performance of A at 2000 episodes. At the end of the experiment, the program performs slightly above 4.5 moves and is close to optimal. For six blocks, the original program averages around 28 moves per problem as shown in Figure 2e. In adaptive mode, this improves to around 58 moves by 100 episodes, and at the end of the experiment to around 10 moves. This is higher than the baseline of 7.1 moves but slightly better than adaptive performance with A that averages around 12 moves in that timeframe. Overall, B performs far better than A due to its informed strategy, and this performance also translates to faster and better learning.

**Program C:** In contrast to the other programs, C is already known to perform close to optimal, and achieves around 4.5 moves on average per problem of four blocks as shown in Figure 2c. With adaptive ordering, this does not seem to improve in the 2000 episodes that we ran the experiment for. This is expected since the program is already performing close to optimal. However, interestingly we know from previous studies that C does not perform optimally for certain “deadlock” cases. We would have hoped to overcome this using learning but from the averaged results this is not evident as there is no significant difference in the performance with and without learning. Importantly, for six blocks, for the first time in the experiments we see that the adaptive ordering actually performs worse than the original program in Figure 2f, albeit by only 0.25 moves per problem on average at its worst. On closer analysis this seems to be because we simply have not run the experiment long enough. Certainly the difference between the two modes of execution is diminishing as the experiment progresses and is evident in Figure 2f. We should note that regardless, the performance of C with or without learning is significantly better than the other programs at around 8 moves and only slightly higher than the baseline case of \(2(n - \sqrt{n}) = 7.1\). Overall, we can conclude that C is already very informed about the domain, so learning is not very useful in this case.

Interestingly, in all experiments, adaptive mode does not do any worse than the default behaviour. This is a useful insight for agent programmers who may otherwise feel reluctant to try a “black box” technology that directly impacts the performance of the agent but that they do not really understand. Another important point is that the performance improvement with adaptive mode is...
very much tied to the problem at hand and does not necessarily generalise to other related problems. For instance, the learning from four-block problems does not generalise to six-block problems and the agent programmer should be aware that one cannot simply plug-and-play learnt values between problems. While this feature may be desirable in many domains, it is nevertheless a shortcoming that comes with the ease of use of the programming model that completely insulates the programmer from the knowledge representation used for learning.

6 Discussion and Conclusion

In this paper we have shown how the mental state representation of an agent program may be exploited to significantly increase the effectiveness of the program through ongoing learning. The novelty is that this performance improvement comes almost for free since the programming model remains relatively unchanged. In particular, we presented an enhancement to the Goal agent programming language that allows adaptive behaviours to be easily programmed. The new language primitive is implemented using a Q-Learning mechanism under the hood, and allows action choices resulting from programmed rules to be improved over time based on the ongoing experience of the agent. A key feature of this enhancement is that it can be readily used by agent programmers who are non-experts in machine learning, since the learning feature has little impact on the programming model. We demonstrated the usability of the framework in the Blocks World domain and analysed the programmer’s role in balancing between fixed and flexible behaviour using three sample solutions for the problem.

The results in Section 5, however, also indicate that scalability (i.e. managing the size of the state space) remains an important challenge. The main tool a programmer currently has in our approach to integrating learning into Goal to reduce the state space is to add and exploit knowledge about the environment in the agent program. Even though the use of domain knowledge may reduce the size of the state space, which corresponds one-to-one with the number of beliefs and goals of the agent, the state space still quickly becomes very large in the Blocks World environment with an increasing number of blocks [33].

We have used and integrated a standard Q-Learning approach to reinforcement learning. It is well-known that such an approach is unable to handle all but the smallest state spaces [14]. Our approach, however, does not depend on this particular choice of learning technique that has been used here mainly to demonstrate the viability of the approach. In order to handle bigger state spaces it is clear that we need some abstraction technique.

The ease of use of the new adaptive functionality in Goal is appealing from a programming point of view as shown in this study. The downside is that a programmer may waste valuable time in trying to improve performance where it is simply not possible within the constraints of the learning framework and the mental state representation used. For example, in a maze world, the only way to distinguish between two T-junctions that “look” identical is to trace back the history of actions that led to the junctions. Here the underlying reinforcement learning framework is inadequate for learning if the mental state only consists of
the current percepts of the agent. Keeping the history in the mental state would help but will make the learning impractical even for simple problems. This drawback also highlights the need for future work to better understand how aware a programmer needs to be of the learning model. It would be useful in this context to develop design patterns that serve as guidelines for implementing adaptive behaviours in typical scenarios. Another avenue for future work is in deciding which mental state atoms are more relevant than others, in order to improve learning times in large state spaces. One option is to automatically learn such useful “features” of the agent’s mental state using regularization techniques [35].

Acknowledgments. This research is supported by the 2011 Endeavour Research Fellowship program of the Australian government.

References

Author Index

Abdel-Naby, S., 69
Alechina, N., 117
Behrens, T., 117
Braubach, L., 23
Broxvall, M., 69
Collier, R., 85
Díaz, A.F., 7
Dastani, M., 39
Dragone, M., 69
Earle, C.B., 7
Fredlund, L., 7
Hindriks, K., 55, 117, 148
Jander, K., 101
Jordan, H.R., 85
Lamersdorf, W., 101
Lillis, D., 85
Logan, B., 117
Meyer, J., 39
Pokahr, A., 23
Ricci, A., 132
Santi, A., 132
Singh, D., 148
Swords, D., 69
Torre, L., 39
Wei, C., 55
Ziafati, P., 39