Current Issues in Multi-Agent Systems Development (Invited Paper)

Rafael H. Bordini[1], Mehdi Dastani[2], and Michael Winikoff[3]*

[1] University of Durham, U.K.
R.Bordini@durham.ac.uk
[2] Utrecht University, The Netherlands
mehdi@cs.uu.nl
[3] RMIT University, Australia
winikoff@cs.rmit.edu.au

Abstract. This paper is based on an invited talk, delivered by the first author, but prepared by all three authors. The paper surveys the state-of-the-art in developing multi-agent systems, and sets out to answer the questions: "what are the key outstanding issues in developing multi-agent systems?" and "what should we, as a research community, be paying particular attention to, over the next few years?". Based on our characterisation of the current state-of-the-art in developing MAS, we identify three key areas for work: techniques for integrating design and code; extending agent-oriented programming languages to cover certain aspects that are currently weak or missing (e.g. social concepts, and modelling the environment); and development of debugging and verification techniques, with a particular focus on using model checking in testing and debugging, and on extending model checking to design artefacts.

1 Introduction

In this paper we survey the current state of the art in multi-agent system development and identify current issues. These issues are areas where we believe the research community should concentrate future research efforts, since they are, in the authors' opinion, crucial to practical adoption and deployment of agent technology.

This paper was based on an invited talk at ESAW 2006. The talk was presented by Rafael Bordini, but the contents of the talk grew out of each of the author's opinions, as presented and discussed at a Dagstuhl Seminar[4], and incorporating ideas from the AgentLink ProMAS technical forum group[5].

This paper is structured as follows. We begin (section 2) by reviewing the current state-of-the-art in MAS development, focussing in particular on Agent Oriented Software Engineering (AOSE), Agent Oriented Programming Languages (AOPLs), and on verification of agent systems. We then (section 3) identify a number of key issues. For

* Michael Winikoff acknowledges the support of the Australian Research Council & Agent Oriented Software (grant LP0453486).

[4] http://www.dagstuhl.de/de/program/calendar/semhp/?semnr=06261

[5] http://www.cs.uu.nl/~mehdi/al3promas.html

each issue we discuss what we believe is the way forward towards resolving the issue. Since this paper is all about future work, our conclusion (section 4) merely summarises the key points of the paper.

2 State of the Art

There are a number of methodologies that provide developers with a process for doing software engineering of multi-agent systems, and there is a range of programming languages especially designed to facilitate the programming of agent system. The field of verification and validation of agent systems is comparatively less well-developed, but there has been work on both formal verification using model checking, and on approaches for debugging and testing agent systems.

In current practice, the way in which a multi-agent system is typically developed is that the developer designs the agent organisation and the individual agents (perhaps using an AOSE methodology), then takes the detailed design and manually codes the agents in some programming language, perhaps agent-oriented, but more often using traditional programming languages. The resulting system is debugged (at best) using a combination of tracing messages and agent inspectors.

Thus developing a multi-agent system, like developing any software system, encompasses activities that are traditionally classified into three broad areas: software engineering (e.g. requirements elicitation, analysis, design[6]), implementation (using some *suitable* programming language), and verification/validation. To help structure this paper, which has also a focus on agent programming languages and verification, we have separated the last two types of activities from general software engineering (specifically analysis and design). Therefore, in the following subsections we briefly review the state of the art in AOSE, programming languages for multi-agent systems, and verification of multi-agent systems (including debugging).

2.1 Agent Oriented Software Engineering

Agent Oriented Software Engineering is concerned with how to do software engineering of agent-oriented systems. It is a relatively youthful field, with the first AOSE workshop held in the year 2000. Nevertheless, over the past decade or so there has been considerable work by many people, resulting in quite a number of methodologies in the literature. These methodologies vary considerably in terms of the level of detail that is provided, the maturity of the methodology, and the availability of both descriptions that are accessible to developers (i.e., not researchers) and of tool support. Of the many methodologies available (e.g. see [6, 31]), key methodologies that are widely regarded as mature include Gaia [63, 66], MaSE [23], Tropos [13] and Prometheus [43].

It is important to clarify what is meant by a *methodology*. From a pragmatic point of view, a methodology needs to include all that a software engineer requires to do analysis and design, namely:

[6] Design includes various sorts of design activities, e.g. architectural design, social design, detailed design.

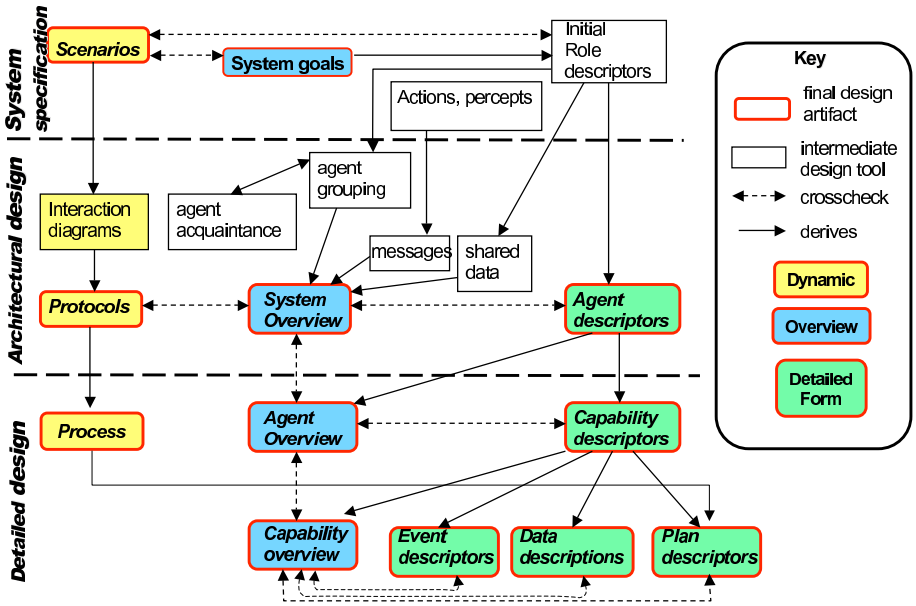

Fig. 1. Prometheus

Concepts: While for object-oriented design the concepts used — classes, objects, inheritance, etc. — are so commonly understood as to be taken for granted, for agents a single set of basic concepts is not (yet) universally accepted or known, so a methodology needs, for completeness, to define a set of basic concepts that are used. For example, the Prometheus[7] methodology uses a set of concepts that are derived from the definition of an agent: action, percept, goal, event, plan, and belief [60].

Process: A methodology needs to provide an overall process that specifies what is done after what. For example, in Prometheus there are three phases — system specification, architectural design, and detailed design — where each phase consists of steps. For example, system specification includes the steps of identifying the system's goals, and of defining the interface between the system and its environment. Although these steps are often most easily described as being sequential, it is usual to recognise that iteration is the norm when doing software analysis/design. The process used by Prometheus is summarised in Figure 1.

Models and Notations: The results of analysis and design are a collection of models, for example a goal overview model or a system overview model. These models are depicted using some notation, often graphical, typically some variation on "boxes and arrows". Figure 2 shows an example model.

Techniques: It is not enough to merely say that, for example, the second step of the methodology is to develop a goal overview diagram. The software designer, espe-

[7] We shall use Prometheus as an example because we are most familiar with it.

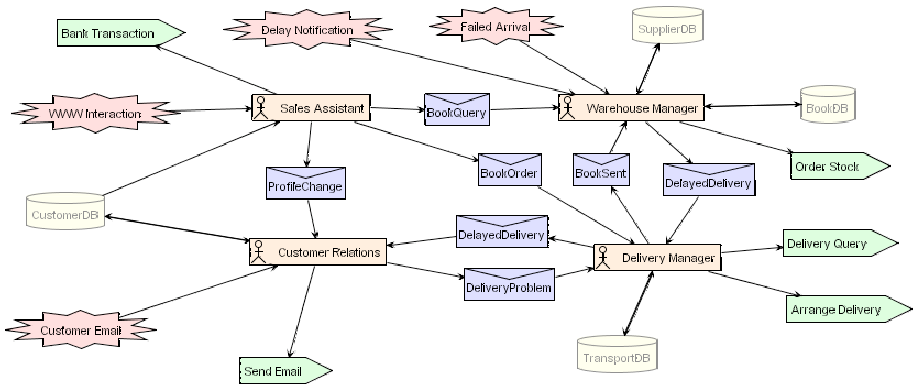

Fig. 2. System Overview Diagram (using Prometheus)

cially if not particularly experienced in designing agent systems, benefits from a collection of specific techniques — usually formulated as heuristics — that guide them in *how* that step is carried out. For example, a goal overview diagram can be refined by asking for each goal "how?" and "why?", yielding respectively new child and new parent goals [53].

Tool Support: Whilst arguably not essential, for any but the smallest design, and for any design that is iteratively refined, having tool support is of enormous benefit. Tools can range from simple drawing packages, to more sophisticates design environments that provide various forms of consistency checking.

Although each methodology has its own processes and notations, there are some common aspects. Many methodologies are broken down into some sort of requirements phase (e.g., "system specification" in Prometheus), some sort of system design (e.g., "architectural design" in Prometheus), and detailed design. The requirements phase specifies what it is that the system should do, with one of the commonly used models being some sort of goal hierarchy. The system design phase determines what agent types exist in the system, and how they interact (i.e., interaction design). Finally, the detailed design phase determines how each agent operates. It is also important to note that many methodologies capture in some way the environment that the agent system will inhabit. In Prometheus this is done, fairly simply, by specifying the interface to the environment in terms of actions and percepts.

2.2 Agent Oriented Programming Languages

There exist many Agent Oriented Programming Languages (AOPLs). In addition, there also exist platforms that focus on providing certain functionalities — such as communication and general infrastructure (e.g., white/yellow pages) — but which do not provide a programming language for developing agents. Such so-called platforms, such

as OAA[15], JADE[5] and FipaOS[47], are not in the scope of this paper as they are not AOPLs.

In this section we shall, instead, focus on agent oriented programming languages for defining the behaviour of individual agents in a multi-agent system. In general, in so-called "cognitive agent programming languages", the focus is on how to describe the behaviour of an agent in terms of constructs such as plans, events, beliefs, goals, and messages. Although there are differences between various proposed AOPLs, they also have significant common characteristics. It is instructive to note that in some of these languages the environment is not captured, that agent interaction is implemented at the level of sending individual message — interaction protocols, for example, are not represented — and that in many languages goals are not provided, but they are approximated by the notion of events instead [61].

Most cognitive agent programming languages such as ***Jason*** [12], Jadex[12], JACK [45], 3APL[22, 32], and 2APL[19] come with their own platforms. These platforms provide general infrastructure (e.g., agent management system, white and yellow pages), a communication layer, and integrated development environment (IDE). The existing IDE's provide editors with syntax highlighting facilities, enable a set of agents programs to be executed in parallel in various modes such as step-by-step or continuous, provide tools to inspect the internal states of the agents during execution, and examine messages that are exchanged by the agents. Further, some of the platforms that come with a cognitive agent programming languages, such as Jadex[46] and 2APL[19], built on an existing agent platform, such as JADE[5]. The resulting platforms use the functionalities of the existing platforms such as the general infrastructure, communication facilities, and inspection tools.

Loosely speaking, agent-oriented programming languages can be classified as being either "theoretical" (i.e., having formal semantics, but arguably being impractical for serious development) or "practical" (i.e., being practical for serious development, but lacking formal semantics). However, it must be noted that this classification is simplistic in that languages with formal semantics are not precluded from being practical. Additionally, there are a number of more recent languages (or extensions of existing languages, such as ***Jason***, which extends AgentSpeak(L)) that aim to be practical, yet have formal semantics. In particular, we mention 2APL, ***Jason***, and SPARK [12, 19, 40] as examples of languages that aim to be practical yet have formal semantics. Below we briefly present some relevant features of ***Jason*** and 2APL. For more information on AOPLs, including various agent programming languages and platforms not mentioned here, see [7, 8].

AgentSpeak(L) and *Jason*

AgentSpeak(L)[8] was proposed by Anand Rao in the mid 90s [50]. Perhaps surprisingly, the proposal received limited attention for a number of years before being "revived": in recent years there have been a number of implementations of AgentSpeak. Of these, the best developed is ***Jason***, which extends AgentSpeak with a range of additional features.

[8] In the remainder of this paper we shall simply refer to this language and its variants as "AgentSpeak".

The AgentSpeak language was intended to capture, in an abstract form, the key execution mechanism of existing Belief-Desire-Intention (BDI) platforms such as PRS [28, 36] and dMARS[24]. This execution cycle links events and plans. Each (relevant) event has a number of plans that it triggers. These plans have so-called context conditions, that specify under what conditions a plan should be considered to be applicable to handle an event. A given event instance is handled by determining which of the plans that it triggers are currently applicable, and then selecting one of these plans and executing it. Execution of plans is done step-by-step in an interleaved manner; plan instances form *intentions*, each representing one of the various foci of attention for the agent. The execution switches to the plan in the focus of attention of currently greatest importance for the agent.

Figure 3 shows examples of AgentSpeak plans for an abstract scenario of a planetary exploration robot. The first plan is triggered when the robot perceives a green patch on a rock, which indicates that rock should be examined first as its analysis is likely to contain important data for scientist working on the mission. In particular, the triggering event is `green_patch(Rock)` which occurs when the agent has a new belief of that form. However, the plan is only to be used when the battery charge is not low (this is the plan *context*, in this case `battery_charge(low)` must not be believed by the agent). The course of action prescribed by this plan, for when the relevant event happens and the context condition is satisfied is as follows: the location of the rock with the perceived green patch is to be retrieved from the belief base (`?location(Rock,Coordinates)`), then the agent should have a new goal to traverse to those coordinates (`!traverse(Coordinates)`), and finally, after the traverse has been successfully achieved, having a new goal to examine that particular rock (`!examine(Rock)`). The other two plans give alternative courses of action for when the robot comes to have a new goal of traversing to a certain coordinate. When the robot believes there is a safe path towards there, all it has to do is to execute the action (that the robot is hardwired to do) of physically moving towards those coordinates. The figure omits for the sake of space the alternative course of action (which would be to move around to try and find a different location from which a safe path can be perceived, probably by using some image processing software).

Jason[9] is a Java-based platform that implements the operational semantics of an extended version of AgentSpeak. The purpose of the language extensions was to turn the abstract AgentSpeak(L) language originally defined by Rao into a practical programming language. The language extensions and the platform have the following features:

Strong negation: as agents typically operate under uncertainty and in dynamic environments, it helps the modelling of such applications if we are able to refer to things agents believe to be true, believe to be false, or are ignorant about.

Handling of plan failures: as multi-agent system operate in unpredictable environment, plans can fail to achieve the goals they were written to achieve. It is therefore important that agent languages provide mechanisms to handle plan failure.

Speech-act based communication: as the mental attitudes that are classically used to give semantics for speech-act based communication are formally defined for

[9] ***Jason*** is jointly developed by Rafael Bordini and Jomi F. Hübner (FURB, Brazil) and available *open source* at http://jason.sf.net.

```
+green_patch(Rock) :
   not battery_charge(low) <-
      ?location(Rock,Coordinates);
      !traverse(Coordinates);
      !examine(Rock).
+!traverse(Coords) :
   safe_path(Coords) <-
      move_towards(Coords).
+!traverse(Coords) :
   not safe_path(Coords) <-
      ...
```

Fig. 3. Examples of AgentSpeak Plans

AgentSpeak we can give precise semantics for how agents should interpret the basic illocutionary forces, and this has been implemented in ***Jason***. An interesting extension of the language is that beliefs can have "annotations" which can be useful for application-specific tasks, but there is one standard annotations that is done automatically by ***Jason***, which is on the *source* of each particular belief.

Plan annotations: in the same way that beliefs can have annotations, programmers can add annotations to plan labels, which can be used by elaborate (e.g., using decision-theoretic techniques) selection functions. Selection functions are user-defined functions which are used by the interpreter, including which plan should be given preference in case various different plans happen to be considered applicable for a particular event.

Distribution: the platform makes it easy to define the agents that will take part in the system and also determine in which machines each will run, if actual distribution is necessary. The infrastructure for actual distribution can be changed (e.g., if a particular application needs to use a particular distribution platform such as JADE).

Environments: multi-agent systems will normally be deployed in some real-world environment. Even in that case, during development a simulation of the environment will be needed. ***Jason*** provides support for developing environments, which are programmed in Java rather than an agent language.

Customisation: programmers can customise two important parts of the agent platform by providing application-specific Java methods: the agent class and the agent architecture (note that the AgentSpeak interpreter provides only the reasoning component of the overall agent architecture). For more details, see [11].

Language extensibility and legacy code: the AgentSpeak extension available with ***Jason*** has a construct called "internal actions". These are then implemented in Java (or indeed any other language using JNI). This provides for straightforward language extensibility, which is also a straightforward way of invoking legacy code from within the high-level agent reasoning in an elegant manner. ***Jason*** comes with a library of essential standard internal actions. These implement a variety of useful

operations for practical programming, but most importantly, they provide the means for programmers to do important things for BDI-inspired programming that were not possible in the original AgentSpeak language, such as checking and dropping the agent's own desires/intentions.

Integrated Development Environment: ***Jason*** is distributed with an IDE which provides a GUI for managing the system's project (the multi-agent system), editing the source code of individual agents, and running the system. Another tool provided as part of the IDE allows the user to inspect agents' internal (i.e., "mental") states when the system is running in debugging mode. The IDE is a plug-in to jEdit (http://www.jedit.org/), and an Eclipse plug-in is likely to be available in the future.

There is also much ongoing research to extend ***Jason*** is various ways, including: plan patterns for declarative goals [34], combination with the Moise+ [35] organisational model (http://moise.sf.net), automated belief revision [1], and combination with a high-level environment language aimed at social simulation, which in recent work aims to allow normative and organisational aspects to be associated with, for example, certain environment locations [41].

2APL: A Practical Agent Programming Language

One of the challenges of practical cognitive agent programming languages is an effective integration of declarative and imperative style programming. The declarative style programming should facilitate the implementation of the mental state of agents allowing agents to reason about their beliefs and goals and update them accordingly. An important issue here is the expressivity of the beliefs and goals, the expressions with which they can be updated, interface to existing declarative languages, and the relation between beliefs and goals (e.g., is it possible for an agent to have an expression as belief and goal at the same time? I.e., can an agent desire a state which is believed to be achieved?) [20, 62]. The imperative style programming should facilitate the implementation of processes, their execution modes, the flow of control, interface to existing imperative programming languages, and processing of events and exceptions. The question is how to integrate these declarative and imperative programming aspects in an effective way. This design objective is the main motivation for introducing a new agent programming language called 2APL (A Practical Agent Programming Language) [19].

Agents that are implemented by 2APL programs can generate plans by reasoning about their goals and beliefs, which are implemented in a declarative way. Plans can consist of actions of different types. Like most BDI-based programming languages, 2APL provides different types of actions such as belief and goal update actions, belief and goal test actions, external actions (to be performed in the agents' shared environment), and communication actions. As agents may operate in dynamic environments, they have to observe (be notified about) their environmental changes. In 2APL such environmental changes will be notified to the agents by means of *events*.

A characterising feature of 2APL is its distinction between events and goals. In some agent programming languages events are used for various purposes, e.g., for modelling an agent's goals or for notifying an agent about internal changes. Although both

goals and events cause a 2APL agent to execute actions, they differ from each other in a principle way. For example, an agent's goal denotes a desirable state for which the agent performs actions to achieve it (goals are directly related with beliefs such that a goal is automatically dropped as soon as it is achieved), while an event carries information about (environmental) changes which may cause an agent to react and execute certain actions. After the execution of actions, an agent's goal may be dropped if the state denoted by it is believed to be achieved, while an event can be dropped just before executing the actions that are triggered by it. Moreover, because of the declarative nature of goals (logical expressions), an agent can reason about its goals while an event only carries information which is not necessarily the subject of reasoning.

```
Beliefs:
   post(5,5).
   dirt(3,6).
   dirt(5,4).
   clean(world) :- not dirt(X,Y).
Goals:
   hasGold(2) and clean(world) ,
   hasGold(5)
PG-rules:
   clean(world) <- dirt(X,Y) |
   {  goto(X,Y);
      PickUpDirt();
      goto(2,2);
      DropDirt() }
PC-rules:
   goldAt(X,Y) <- true | { [goto(X,Y); PickUpGold()] }
PR-rules:
   PickUpDirt();R <- dirt(X,Y) | { goto(X,Y);PickUpDirt();R }
```

Fig. 4. Examples of 2APL Program

For example, consider the 2APL program as illustrated in Figure 4. This program, which for simplicity reasons does not include all details, indicates that the agent starts with the beliefs that it is on position (5,5), there are dirts at positions (3,6) and (5,4), and that the world is clean if there is no dirt at any position. The agent wants to achieve two states (the goals are separated by a comma): one state in which he has two pieces of gold *and* the world is clean of dirt, and another state in which he has five pieces of gold. Note that these two state does not need to be achieved simultaneously. The planning goals rule (PG-rules) indicates that the state in which the world is clean can be achieved by going to the dirts' positions, picking them up, bringing them to the depot position (2,2), and dropping them in the depot. Note that the application of this rule

can only achieve the subgoal `clean(world)`, not the desired state `hasGold(2) and clean(world)`. The ability to achieve subgoals requires reasoning about goals. Different notions of reasoning about goals are discussed in [20, 54]. Note also that if all dirts are picked up and dropped in the depot position, then the agent will believe that the world is clean. If the agent also believes that it has two pieces of gold, then it automatically drops the goal `hasGold(2) and clean(world)`.

The difference between goals and events can be illustrated by the procedural rules (PC-rules). This rule indicates that if the agent is notified by an event that there is a gold piece at a certain position, then the agent should go to that position and pick up the gold piece. Note that both goals and event can cause the agent to perform actions.

Other characterising 2APL features are related to the constructs designed with respect to an agent's plans. The first construct is a part of an exception handling mechanism allowing a programmer to specify how an agent should repair its plans when the execution of its plans fail. This construct has the form of a rule which indicates that a plan should be replaced by another one. For example, consider again the agent program illustrated in Figure 4. The plan repair rule (PR-rules) indicates that if the execution of a plan that starts with the action `PickUpDirt()` (followed by the rest `R` of the plan) fails (for example because the dirt was already removed by another agent), then the plan should be replaced by the `goto(X,Y);PickUpDirt();R` plan if the agent believes that there is dirt at another position (X,Y). Note the use of variable `R` which stands for the rest of the original plan. The second 2APL programming construct related to plans is the so-called non-interleaving (region of) plans. In most agent-oriented programming languages, an agent can have a set of plans whose executions can be interleaved. The arbitrary interleaving of plans may be problematic in some cases such that a programmer may want to indicate that a certain part of a plan should be executed at once without being interleaved with the actions of other plans. A non-interleaving (region of) plan can be marked by putting the (region of) plan within `[]` brackets. For example, in Figure 4 the plan for picking up a gold piece when notified by an event is a non-interleaving plan.

In addition to these features, 2APL provides specific programming mechanisms such as procedures, recursion and encapsulation. A procedure can be implemented by means of a specific rule that relates an abstract action (procedure call) to a concrete plan (procedure body). A recursion can be implemented simply by including the procedure call in the procedure body. Although these mechanisms can be implemented in other cognitive agent programming languages, 2APL follows the separation of concerns principle and provides specific constructs for the purpose of implementing procedures and recursions. In comparable cognitive agent programming languages (programming languages with formal semantics), procedures can be implemented by means of rules that relate events (or goals) to plans. In 2APL, procedures and recursion are considered as inherently different concepts from goals and events such that their implementation is independent of these concepts.

2.3 Verification and Validation

Multi-agent systems are distributed and concurrent, and the agents that make up a MAS are able to exhibit complex flexible behaviour in order to achieve its objectives in the

face of a dynamic and uncertain environment. This flexible behaviour is key in making agent technology useful, but it makes it difficult to debug agent systems, and, once the system is (supposedly) debugged and ready for deployment, makes it hard to obtain confidence that the system will work as desired.

Debugging is an essential part of the process of developing software, and so good support for debugging is important. In the case of agent systems, there has been some work on debugging (e.g. [26, 44]), but debugging techniques used in practice still rely on tracing and state inspection. The better agent development environments provide facilities to view, browse, and analyse the messages that are being exchanged, and facilities to examine the internal state of the agents. As examples, Figures 5 and 6 show the *Mind Inspector* tool provided by ***Jason*** and 2APL's State Trace, respectively; other platforms, such as JACK, provide similar functionality.

In addition to using standard debugging techniques, there has been some work that aims to provide semi-automatic bug detection. For example, the work of Poutakidis et al. [44, 48, 49] automatically detects bugs in agent interactions by comparing an execution trace with the interaction protocol that is supposed to describe the valid message sequences. Any sequence of messages that occurs in the system's execution but that is illegal according to the protocol is automatically identified and flagged as an error. The general principle is that design artefacts can be used to assist in debugging.

In any software system it is essential that when the system is deployed and used, there is confidence that it will do what it is supposed to do. Typically, this confidence is achieved through testing. However, for agents that are able to exhibit flexible behaviour, achieving their goals in a range of ways depending on the situation, it is harder to achieve confidence in the system through testing. Hence, there has been a rather limited amount of work on testing agent systems, but there has been interest in using formal methods, especially model checking, to verify agent systems.

Work has focussed on model checking because it is easier to use than theorem proving, and, more importantly, because it can provide counter examples when the system fails to satisfy a desired property. Further, much work is devoted to state-space reduction techniques which can make model checking practical even for very large systems. However, although the technology is promising, at present it is fairly preliminary: the languages handled are limited, and the techniques have not been applied to industrial-scale case studies in multi-agent system.

Another type of work related to the correct behaviour of agent programs aims at specifying the semantics of the agent programming language is such a way to guarantee the satisfaction of certain behaviour. For example, in [21] it was shown that the semantics of an agent programming language can be defined in such a way that any agent implemented in that agent programming language will drop its goals if the goal is either achieved or believed not to be achievable anymore.

Most of the work done on model checking within the multi-agent systems research area is quite theoretical, although there are approaches that use existing model checkers, typically to check properties of particular aspects of a multi-agent system. A survey paper on the use of logic-based techniques for specifying but particular for *verifying* multi-agent systems is to appear in print around the same time as this paper, so instead of giving references here, we refer the interested reader to [25]. When it comes to model

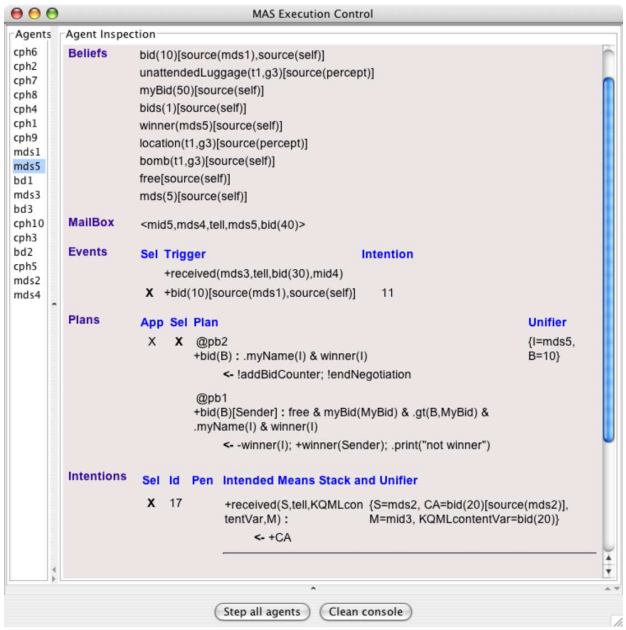

Fig. 5. ***Jason*** Debugging: Agent Mind Inspector

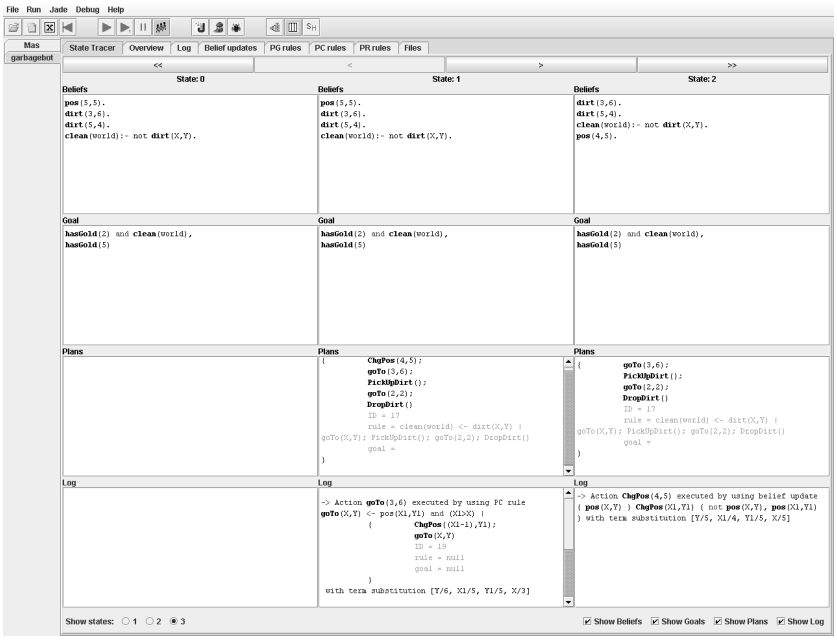

Fig. 6. 2APL Debugging: Agent State Tracer

checking software (i.e., a complete running system) there is little work that applies to multi-agent systems in particular. More specifically on model checking agent *programs* written in an agent-oriented programming language, to our knowledge the only existing approach is the one presented in [10, 55].

3 Problems with the Current State of the Art

Let us briefly summarise the current state-of-the-art in developing MAS. Not all current multi-agent system development projects use all of these, but rather we describe what is already available and in our opinion is likely to be used. Indeed adoption of these techniques will make short-term future development of agent-based system far more successful than previous attempts, in our opinion. We then discuss how this process should be improved by future research. We consider the state-of-the-art development process to be as follows:

1. Designing organisation and individual agents *using an AOSE methodology*
2. Taking the resulting design and (manually) coding the agents *in some AOPL*, based on the design
3. Debugging the system using message tracing and agent inspectors
4. Possibly using model checking on agent code (but unlikely)

Even though we believe that adoption of this development process would already improve significantly the development of multi-agent systems, the above summary highlights a number of areas where the current state-of-the-art is, in our opinion, seriously lacking, and where future work is sorely needed. There are three key issues:

- The implementation is developed completely *manually* from the design. This creates the possibility for the design and implementation to diverge, which tends to make the design less useful for further work in maintenance and comprehension of the system.
- Although present AOPLs provide powerful features for specifying the internals of a single agent, they mostly[10] only provide messages as the mechanism for agent interaction. Messages are really just the least common denominator for interaction, and, especially if flexible and robust agent interactions are desired, it is important to design and implement agent interactions in terms of higher-level concepts such as social commitments [27, 64], delegation of goal/task, responsibility[29, 30], or interaction goals [14]. Additionally, AOPLs are weak in allowing the developer to model the environment within which the agents will execute.
- In most of the practical approaches for verification of multi-agent systems, verification is done on code. While this has the advantage of proving properties of the system that will be actually deployed, it is also often useful to check properties during the system design, so more work is required in verification of agent design artefacts. In fact, all the work on model checking for multi-agent systems is still in early stages so not really suitable for use on large and realistic systems.

[10] Although there has been work on AOPL support for programming *teams* of agents (e.g. [17, 33, 52]), this approach only applies to certain problem domains, where agents are co-operative.

In the remainder of this section we tackle each of these issues, and describe where we believe we should be heading, and what we believe needs to be done to address each of these issues. In brief, we believe that the key things we, as a research community, should be doing with respect to these issues are:

- Working on developing techniques and tools that allow for designs and code to be strongly integrated with consistency checking and change propagation.
- Developing better integrated designs and code would be facilitated by AOPLs being closer to the design in terms of covered concepts — while this is already true for individual agent abstractions, that is not the case for social abstractions. Thus, we believe that research effort in AOPLs should in the short-term concentrate on extending AOPLs so they cover design concepts that are presently either missing or not covered well. Such concepts include interaction concepts at a higher level than messages (e.g., interaction protocols, social commitments, norms, obligations, responsibility, trust), and the environment (e.g., resources, services, actions), although further work on certain types of declarative goals is still required [18].
- Develop better techniques and tools for debugging and verification. One approach that is enabled by the existence of design that is reliably consistent with the code[11] is to use design artefacts to assist with debugging (e.g., [44]). However, debugging alone cannot assure us of the correctness of a system, and so formal verification techniques are also important. Interestingly, formal verification techniques such as model checking can be used to help validation when formal verification turns out not to be possible in practice (e.g., [57]).

3.1 Integrating Code and Design

There is an unfortunate tendency in the computing world to regard design and code as being completely different beasts. There are some clear differences between them: for instance, code is usually textual and detail-rich, whereas design is usually graphical and high-level. However, as was lucidly argued as far back as 1992 "Programming is a design activity" [51]. That is, the programming process, which is often (incorrectly) related by analogy to manufacturing a design in other engineering disciplines, is in fact a design activity, which is why it involves considerable rework in the form of debugging. Thus, it is highly desirable to have code and design being seen as different views on what is really a single conceptual activity.

Unfortunately, the current state-of-the-art in linking design and code is surprisingly primitive: "In most cases, the reverse-engineering facilities provided by CASE-tools supporting the Unified Modelling Language (UML) are limited to class diagram extraction" [38].

In an attempt to be systematic, we briefly present a taxonomy of the possible approaches for eliminating the "gap" between code and design. We have identified eight possible approaches:

[11] In fact, using design artefacts for debugging can also assist in detecting inconsistencies between design and code.

Eliminate design: one way of avoiding discrepancies between two entities is to eliminate one of them! By "eliminating design" we do not mean that design activities are not performed, but that the results of these activities (in the form of design artefacts) are not retained and maintained. This approach, which may sound impossibly naive, is in fact what agile methodologies such as XP [4] propose. This approach is feasible if the application's design is relatively simple and/or is familiar to the system's developers.

Eliminate code: instead of eliminating design, we could eliminate code. Clearly, in order to have running software we need to augment the design with additional details. This approach corresponds to Model-Driven Development (MDD). This has been shown to be practical in certain cases, but has the drawback that the design can become cluttered with the additional details needed to make it executable.

Generate code from design: a third approach is to generate the code from the design. This can be done fairly easily (although usually there is not enough information in the design to generate more than skeleton code). However, without additional techniques to then ensure the continued consistency of design and code as one or the other is changed, this is not a useful solution.

Extract design from code (reverse engineering): this automation possibility is intriguing, but not practical yet. Also, code typically does not contain all desired design information. However, the code can be extended to encompass such information.

Extract changes from design and apply to code: there is an issue here with language-dependence; that is, tools need to be developed for the particular design notation and the target programming language so that changes in the design can be reflected in the right way for that programming language. Also, it does not actually solve the problem (in case the code is changed directly)!

Extract changes from code and apply to design: the same issue with language-dependence as in the item above exists here. Also, it does not actually solve problem (in case the design gets changed)!

Extract changes in design/code and apply to the other: although this approach completely solves the problem, the issue of language dependence still remains.

Integrate code and design into a single model: in this approach design and code become just different views on an underlying model which encompasses both. This avoids problem with language dependence by committing to a given programming language for the methodology, but requires integration between design and programming tools.

An issue in integrating code and design is that there are many design notations (and associated tools), and many AOPLs. Having to develop a link between each possible design tool and each possible language is clearly undesirable. A naive solution to this problem would be firstly to get the AOSE research community to agree on a single methodology and come together to develop a single support tool, and then secondly to get the AOPL research community to agree on a single AOPL. Clearly, this is not something that is likely to happen any time soon!

A more complex, but far more practical approach is to standardise interchange formats and APIs, while allowing the underlying notations/languages/tools to remain diverse. This is the approach we believe is most suitable for the multi-agent systems community and therefore we propose that the research community:

– Develop a standard abstraction for AOPLs
– Develop a standard API for making changes to an agent program (cf. Jadex ADF [46])
– For each AOPL's implementation, an implementation of the API is created
– Each design tool is extended with the ability to push changes into code via the API.

and, symmetrically, it is also required that the community:

– Agree on a common set of design abstractions
– Develop a standard API for making changes to a multi-agent system design
– For each AOSE methodology, an implementation of the API is created
– Each programming tool is extended with the ability to push changes into design via the API.

Clearly this will require major research effort and collaboration within (and between) the AOSE and ProMAS communities, but we believe this will have a significant impact in future MAS development, and that this line of research is worth pursuing.

3.2 Extending Agent-Oriented Programming Languages with Organisation and Interaction Aspects

Organisations are useful because they allow us to address at design and runtime how a complex multi-agent system should behave. Concepts such as responsibility, power, task delegation, norms, role enactment, workflows, shared goals, access control, groups and social structure help a software developer to understand and implement large multi-agent systems. Agent development methodologies need to provide concepts to specify, design, and implement static and dynamic aspects of such organisations. Though concepts for static views of organisations appear in almost all methodologies, the dynamics are not widely and thoroughly considered yet.

Historically, agent-oriented programming languages have focused on the *internals* of agents and have somewhat neglected *social* and organisational aspects. Most existing agent programming languages do not provide programming constructs to implement such multi-agent aspects so that programmers have to translate and incorporate these features at the level of individual agents' internals. However, some existing programming languages allow the implementation of these aspects, although to a very limited extent. For example, ***Jason*** provides programming constructs to indicate the infrastructure to be used by the agents, the environment the agent will share, and the agents and their numbers to be created and executed. Also, 2APL provides programming constructs to indicate which individual agents and how many of them should be created, and which agent has access to which environment.

One reason for neglecting these issues is the lack of clear and computational semantics for these social and organisational concepts. A starting point to tackle this issue is to develop formal and computational semantics for social and organisational concepts based on theories of concurrency and coordination, and possibly inspiration from theories of human organisations. It should be noted that work on electronic institutions, such as Islander, which maps to an implementation platform called Ameli [2], regulates agent interactions and ensures that the laws of the institution are obeyed. Although work

in that area does not focus on designing agent programming languages, they can be a source of inspiration for designing agent programming languages with specific programming constructs that allow the implementation of multi-agent organisations and interactions organisations.

Extending AOPLs: Interaction

Current agent-oriented programming languages allow the implementation of agent interactions at the level of messages. It is, however, desirable to move beyond messages because designing and implementing at message level gives "brittle" interactions. Also, designing and implementing at this level makes it very hard to verify/debug and modify the interaction between agents. In order to overcome these problems, the following options can be considered.

One can extend agent-oriented programming languages with programming constructs that enable the implementation of interaction protocols. The execution of implemented protocols should enable individual agents to perform appropriate actions to achieve desirable interactions when they so choose. Alternatively, the execution of these programming constructs could extend the individual agent programs with the appropriate actions such that the execution of extended agent programs guarantee the desirable interactions between the agents.

Other options include using alternatives to message-centric interaction protocols, such as specifying interactions in terms of social commitments (e.g. [27, 65]), landmarks [39] or interaction goals [14]; and extending AOPLs with support for these concepts [59].

Extending AOPLs: Environments

The environment shared by agents can be seen as a first-class abstraction which is as important as agents [58]. It provides the surrounding conditions for agents to exist and contains elements that are not present in agents, which are often important means for agent interaction. The environment can be used to help building a solution (coordination marks, such as pheromones, are a typical example). Agents can influence the environment to make it change or to extract meaningful information (perception). Agents can also communicate indirectly via the environment by adding and reading information from the environment. Finally, from the decision theory point of view, an agent decides which action to perform in an environment while the environment determines the actual effects of the action.

The environment benefits agent technology because it contributes to the separation of concerns and forces designers to incorporate appropriate agent features. Environments can be considered as a set of artefacts [42] used by agents to achieve goals and that regulate agent interaction. More elaborate approaches try to give more explicit definitions of environments by defining a framework. This framework would be responsible for executing agent actions and determining the effects of such actions.

Some agent development methodologies such as Prometheus generate agent system designs that include a primitive environment model. The environment model is captured as *actions* and *percepts*. Also, some of the existing agent programming languages

such as 3APL, 2APL, and ***Jason*** support the implementation of external shared environments. These environment are implemented as Java classes, for example so that their methods correspond with the actions that agents can perform in the environment. The state of the environment is then implemented by class variables which will be changed by the agents' actions (method calls). The modification of the state of the environment is implemented by the methods of the class. It is important to emphasis that these agent programming languages use Java to implement the agents' shared environment. Future work should consider extending the existing agent programming languages with specific abstract programming constructs that facilitate implementation of the environment in terms of high-level concepts such as resource, service, actions, and action effects.

3.3 Verification and Validation

Some of the reasons why debugging MAS is so hard are: agents exhibit *flexible* behaviour so it may turn out to be difficult to detect the circumstance that led to the faulty behaviour and even more so to fix the problems in a way that is consistent in all behaviours; the inherent *concurrency* in the system is an obvious complication as concurrent systems are notoriously difficult to debug; the environment is typically failure-prone so it may again be difficult to detect/reproduce the exact circumstances that cause problems and ensure that it will work correctly in the future; there will be typically a large number of agents which clearly makes things more difficult; systems might be open, so possibly difficult to consider the consequence of changes for various combinations of participating agents; each agent has a complex mental structure which needs to be not only inspected, but also understood; there will be typically a large number of communication messages that might need to be analysed in conjunction with agents' mental states. More importantly, the whole process needs to be tailored to account for the high-level notions used in MAS, such as beliefs, goals, plans, norms, roles, groups, etc. We strongly expect a lot of research to be done in this area to produce debugging approaches and tools which address these and many other specific issues in debugging multi-agent systems.

Whilst debugging and testing are fundamental, some applications require more than that. Many applications of multi-agent systems need to be *dependable systems*. Ideally, we would like to be able to fully *verify*, using formal methods, a multi-agent system which is safety/business/mission-critical. A popular current approach for formal verification of software is *model checking* [16]. Unfortunately, model checking techniques for verification of agent systems is still in its infancy (particularly in regards to practical tools). We expect to see a lot of work being done also in this area, and indeed there is already an active research community with ongoing projects in this direction.

In summary, in order to provide good support for ensuring that MAS run correctly, much work is needed in testing, debugging, and verification approaches and tools. More interestingly, approaches that combine these three activities are also likely to emerge for MAS, as they have in the automated software verification literature with approaches for traditional systems/languages. For example, when full verification is not possible because the system's state space is too large even after the use of state-space reduction techniques, practical model checking tools can still be used, for example, to help find

the required input leading to special cases that can be potentially useful in testing a system [57].

One possible direction for research on model-checking multi-agent systems is as follows. In the same way that there is work being done for model checking to be applied to systems programmed in agent-oriented programming languages, it would be interesting to see approaches that apply directly to design documents of AOSE methodologies. This would, however, require that the design notations are given formal semantics, which is another interesting research strand. Both approaches exist in model checking traditional software. The idea in model checking programs [56] is to verify the system as it will be run by the users. Because code is much more detailed than design, the state-space explosion problems is normally much worse for programs than for design. In this approach, the use of state-space reduction techniques is particularly important. In fact, much work is done by the automated verification community on that topic, and a variety of different techniques exists for various types of state-space reduction used in model checking (e.g., data abstraction, partial-order reduction, property-based slicing, and compositional reasoning). The advantage of model checking programs is that if we succeed in the verification exercise, we know that the system *as actually run* satisfies the checked properties. When model-checking a high-level description of the system, we need to ensure that no errors are introduced in the implementation, which is typically done by a process of "refinement".

In general terms, what we would like to see in the future, ideally, are model checking techniques that are tailored particularly for MAS; that is, taking into account important agent abstractions such as goals, coalitions, etc. It must be noted though, that certain characteristics of MAS might prove to be particularly difficult to deal with for model checking, such as openness, emergence, etc. On the other hand, it is also possible that characteristics that are typical of MAS specifically can be explored for more efficient verification than normally expected in traditional software (e.g., compositional reasoning may turn out to work particularly well for agent organisations), but at the moment this is at best a conjecture.

One issue is that work on verifying agent programs has been done in the context of a given agent-oriented programming language. Clearly, it is desirable to have model checking tools that can be used on programs written in a range of languages. One approach to doing this is currently being pursued by Fisher, Wooldridge, Bordini, and colleagues. They aim to develop an "Agent Infrastructure Layer" (AIL) in the form of a Java library. There will then be provably correct translation of various programming languages into that Java library, so that JPF [56](http://javapathfinder.sf.net) can be used as a model checker. This would extend the previous approach so that it would apply to a variety of agent programming languages rather than only AgentSpeak. The development of AIL itself should be of interest as it would highlight common aspects of existing programming languages for multi-agent systems. For up-to-date information on that project, see http://www.csc.liv.ac.uk/~michael/mcapl06.html.

Another important area for future research is devising *state-space reduction techniques* specifically created for MAS. As mentioned earlier, much work in the automated verification community centres on state-space reduction techniques, and indeed they are responsible for the (relatively recent) success and popularity of model checking tech-

niques. The availability of such techniques would have a major impact in the scale of multi-agent systems that will be verifiable in practice. Unfortunately, almost no work has yet been done in this direction; some (initial) work in this direction was done in [9], where a property-based slicing technique for an agent language was presented.

Still, we cannot be sure to be able to verify all multi-agent systems, but we would like to emphasise that there is much to be explored in the use of model checkers besides verification (e.g., for debugging and testing). We expect to see work in that direction and we would like to see, eventually, off-the-shelf MAS algorithms with verified properties available for MAS practitioners. That way, even when verification of the whole system is not possible, we would be able to know properties of particulars techniques used in parts of the system, which may turn out to be of great usefulness in many applications, and have such techniques immediately available for reuse by MAS developers.

4 Conclusion

We have surveyed the state-of-the-art in developing multi-agent systems, focussing on the three core areas of software engineering (analysis, design), programming, and verification (including both debugging and formal verification). Based on this we have identified three key areas where we feel the state-of-the-art is lacking, and could be improved. Specifically we believe that there is a need to:

- integrate design and code better;
- extend AOPLs with the ability to represent social aspects and the environment; and
- develop practical tools for verification and validation that are tailored specifically for multi-agent systems.

For each of these three key topics we discussed ways of meeting the challenges, and suggested some possible directions for the agents research community.

References

1. N. Alechina, R. H. Bordini, J. F. Hübner, M. Jago, and B. Logan. Automating belief revision for agentspeak. In M. Baldoni and U. Endriss, editors, *Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies (DALT 2006), held with AAMAS 2006, 8th May, Hakodate, Japan*, pages 1–16, 2006.
2. J. L. Arcos, M. Esteva, P. Noriega, J. A. Rodríguez, and C. Sierra. Engineering open environments with electronic institutions. *Journal on Engineering Applications of Artificial Intelligence*, 18(2):191–204, 2005.
3. G. S. Avrunin and G. Rothermel, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. ACM, 2004.
4. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. ISBN 201-61641-6.
5. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade - a java agent development framework. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 5. Springer-Verlag, 2005.

6. F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems*. Kluwer Academic Publishing (New York), 2004.
7. R. Bordini, L. Braubach, M. Dastani, A. Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.
8. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-agent Programming: Languages, Platforms, and Applications*. Springer, 2005.
9. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19–23 July*, pages 896–903, New York, NY, 2004. ACM Press.
10. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, Mar 2006.
11. R. H. Bordini, J. F. Hübner, et al. ***Jason****: A Java-based interpreter for an extended version of AgentSpeak*, manual, release 0.9 edition, Jul 2006. `http://jason.sourceforge.net/`.
12. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 1, pages 3–37. Springer-Verlag, 2005.
13. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8:203–236, May 2004.
14. C. Cheong and M. Winikoff. Hermes: Designing goal-oriented agent interactions. In *Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE-2005)*, July 2005.
15. A. Cheyer and D. Martin. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.
16. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
17. P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.
18. M. Dastani and J. Gomez-Sanz. Programming multi-agent systems. *The Knowledge Engineering Review*, 20(2):151–164, 2006.
19. M. Dastani, D. Hobo, , and J.-J. C. Meyer. Practical extensions in agent programming languages. In *Proceedings of the sixth International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'07)*. ACM Press, 2007.
20. M. Dastani, M. van Riemsdijk, and J.-J. Meyer. Goal types in agent programming. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06).*, 2006.
21. M. Dastani, M. B. van Riemsdijk, , and J.-J. C. Meyer. On the relation between agent specification and agent programming languages. In *Proceedings of the sixth International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'07)*. ACM Press, 2007.
22. M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer. A programming language for cognitive agents: goal directed 3APL. In M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNCS*, pages 111–130, Berlin, 2004. Springer Verlag.
23. S. A. DeLoach. Analysis and design using MaSE and agentTool. In *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.

24. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. Technical Note 72, Australian Artificial Intelligence Institute, 1997.
25. M. Fisher, R. H. Bordini, B. Hirsch, and P. Torroni. Computational logics and agents: a roadmap of current technologies and future trends. *Computational Intelligence Journal*, 2007. To appear.
26. D. Flater. Debugging agent interactions: a case study. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC2001)*, pages 107–114. ACM Press, 2001.
27. R. A. Flores and R. C. Kremer. A pragmatic approach to build conversation protocols using social commitments. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1242–1243. ACM Press, 2004.
28. M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.
29. D. Grossi, F. Dignum, M. Dastani, and L. Royakkers. Foundations of organizational structures in multi-agent systems. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'05)*. ACM Press, 2005.
30. D. Grossi, F. Dignum, V. Dignum, M. Dastani, and L. Royakkers. Structural aspects of the evaluation of agent organizations. In *Pre-proceedings of COIN@ECAI'06*, 2006.
31. B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group Publishing, 2005.
32. K. V. Hindriks, F. S. D. Boer, W. V. der Hoek, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
33. A. Hodgson, R. Rönnquist, and P. Busetta. Specification of coordinated agent behaviour (the simple team approach). Technical Report 5, Agent Oriented Software, Pty. Ltd., 1999. Available from *http://www.agent-software.com*.
34. J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In M. Baldoni and U. Endriss, editors, *Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies (DALT 2006), held with AAMAS 2006, 8th May, Hakodate, Japan*, pages 65–81, 2006.
35. J. F. Hübner, J. S. Sichman, and O. Boissier. Using the $\mathcal{M}$oise+ for a cooperative framework of MAS reorganisation. In A. L. C. Bazzan and S. Labidi, editors, *Advances in Artificial Intelligence - SBIA 2004, 17th Brazilian Symposium on Artificial Intelligence, São Luis, Maranhão, Brazil, September 29 - October 1, 2004, Proceedings*, volume 3171 of *Lecture Notes in Computer Science*, pages 506–515. Springer, 2004.
36. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
37. N. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors. *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*. IEEE Computer Society, 2004.
38. R. Kollman, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Ninth Working Conference on Reverse Engineering (WCRE'02)*, 2002.
39. S. Kumar, M. J. Huber, and P. R. Cohen. Representing and executing protocols as joint actions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 543 – 550, Bologna, Italy, 15 – 19 July 2002. ACM Press.
40. D. Morley and K. L. Myers. The spark agent framework. In Jennings et al. [37], pages 714–721.
41. F. Y. Okuyama, R. H. Bordini, and A. C. da Rocha Costa. Spatially distributed normative objects. In G. Boella, O. Boissier, E. Matson, and J. Vázquez-Salceda, editors, *Proceedings of the Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN), held with ECAI 2006, 28th August, Riva del Garda, Italy*, 2006.

42. A. Omicini, A. Ricci, and M. Viroli. Coordination artifacts as first-class abstractions for MAS engineering: State of the research. In A. F. Garcia, R. Choren, C. Lucena, P. Giorgini, T. Holvoet, and A. Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, volume 3914 of *LNAI*, pages 71–90. Springer, Apr. 2006. Invited Paper.
43. L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004. ISBN 0-470-86120-7.
44. L. Padgham, M. Winikoff, and D. Poutakidis. Adding debugging support to the Prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173–190, 2005. Special issue on Agent-oriented Software Development.
45. M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, 2001.
46. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 6, pages 149–174. Springer, 2005.
47. S. Poslad, P. Buckle, and R. Hadingham. The fipa-os agent platform: Open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, pages 355–368, 2000.
48. D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02)*, 2002.
49. D. Poutakidis, L. Padgham, and M. Winikoff. An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 628–632, Maebashi City, Japan, 2003.
50. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, 1996. LNAI, Volume 1038.
51. J. Reeves. What is software design? *C++ Journal*, 1992.
52. M. Tambe. Agent architectures for flexible, practical teamwork. In *National Conference on Artificial Intelligence (AAAI-97)*, 1997.
53. A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 249–263, Toronto, 2001.
54. M. B. van Riemsdijk, M. Dastani, , and J.-J. C. Meyer. Subgoal semantics in agent programming. In *Proceedings of 12th Portuguese Conference on Artificial Intelligence*, volume 3808 of *Lecture Notes in Computer Science*, pages 548 – 559. Springer, 2005.
55. M. B. van Riemsdijk, F. de Boer, M. Dastani, , and J.-J. C. Meyer. Prototyping 3apl in the maude term rewriting language. In K. Inoue, K. Satoh, and F. Toni, editors, *Proceedings of the seventh International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VII)*, volume 4371 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.
56. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pages 3–12. IEEE Computer Society, 2000.
57. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In Avrunin and Rothermel [3], pages 97–107.
58. D. Weyns, H. V. D. Parunak, and F. Michel, editors. *Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*. Springer, 2006.

59. M. Winikoff. Implementing commitment-based interactions. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2007.
60. M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 555–568. Springer, LNAI 2256, 2001.
61. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002.
62. L. Winkelhagen, M. Dastani, and J. Broersen. Beliefs in agent implementation. In *Proceedings of the third International Workshop on Declarative Agent Languages and Technologies (DALT 2005)*, volume 3904 of *Lecture Notes in Computer Science*. Springer, 2006.
63. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.
64. P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, 2002.
65. P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence (AMAI), Special Issue on Computational Logic in Multi-Agent Systems*, 2004.
66. F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.