# AgentSpeak(XL):

## Efficient Intention Selection in BDI Agents via Decision-Theoretic Task Scheduling[*]

Rafael H. Bordini[†], Ana L.C. Bazzan,
Rafael de O. Jannone, Daniel M. Basso,
Rosa M. Vicari
Informatics Institute
Federal University of Rio Grande do Sul
91501-970, Porto Alegre-RS, Brazil
{bordini,bazzan,jannone,dmbasso,rosa}@inf.ufrgs.br

Victor R. Lesser
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
lesser@cs.umass.edu

## ABSTRACT

This paper shows how to use a decision-theoretic task scheduler in order to automatically generate efficient intention selection functions for BDI agent-oriented programming languages. We concentrate here on the particular extensions to a known BDI language called AgentSpeak(L) and its interpreter which were necessary so that the integration with a task scheduler was possible. The proposed language, called AgentSpeak(XL), has several other features which increase its usability; some of these are indicated briefly in this paper. We assess the extended language and its interpreter by means of a factory plant scenario where there is one mobile robot that is in charge of packing and storing items, besides other administrative and security tasks. This case study and its simulation results show that, in comparison to AgentSpeak(L), AgentSpeak(XL) provides much easier and efficient implementation of applications that require quantitative reasoning, or require specific control over intentions (e.g., for giving priority to certain tasks once they become intended).

## Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

## General Terms

Languages

## Keywords

BDI programming languages, intention selection, decision-theoretic scheduling

[*]This work has been supported by CNPq and the NSF (grant number IIS-9988784).

[†]Currently at the Department of Computer Science, University of Liverpool, U.K.

## 1. INTRODUCTION

In the course of attempting to develop a powerful and innovative agent framework based on a two-level agent architecture combining cognitive and decision-theoretic frameworks, several ideas have occurred to us which achieve that goal to various degrees. This paper presents a very practical one, which significantly contributes to the performance of the BDI framework we have been developing based on Rao's AgentSpeak(L) agent oriented programming language [10]. Such development relied on our previous experience with a prototype AgentSpeak(L) interpreter called SIM_Speak [7] (to the best of our knowledge, the first working AgentSpeak(L) interpreter), which runs on Sloman's SIM_AGENT toolkit, a testbed for "cognitively rich" agent architectures [12].

The referred integration idea is to use TÆMS [1] and the Design-To-Criteria (DTC) scheduler [14] (see [6] for an overview of that approach to multi-agent systems) to improve the performance of BDI programming languages, in particular concerning issues such as intention selection, on which we concentrate in this paper. We present here AgentSpeak(XL), an extension we have proposed to AgentSpeak(L) for improving that language in various aspects and in particular to accommodate the on-the-fly use of DTC for generating efficient intention selection functions. This has greatly improved the expressiveness of the language, facilitating the programming of certain types of applications (where quantitative reasoning is natural). Also, it has allowed a certain control over an agent's set of intentions which was not possible with the original AgentSpeak(L) interpreter.

The remainder of this paper is structured as follows. The next section provides the necessary background on AgentSpeak(L) and TÆMS/DTC. Section 3 presents the extensions in AgentSpeak(XL) which make it possible to control intention selection by means of high level constructs. We show the advantages of the extended interpreter by means of a simple case study on a factory plant robot given in Section 4. Section 5 briefly describes how to specify that robot's reasoning both in AgentSpeak(L) and AgentSpeak(XL), and it provides quantitative results showing the improvement on the robot's performance achieved in that case study by our AgentSpeak(XL) interpreter. Finally, we mention some future work on alternative ways to handle the integration with decision-theoretic task scheduling, as well as on directions of research pointing out to our long-term goal of integrating cognitive and utilitarian approaches to Multi-Agent Systems.

## 2. BACKGROUND

### 2.1 AgentSpeak(L)

In [10], Rao introduced the AgentSpeak(L) programming language. In that paper, not only has he defined the operation of an abstract interpreter for it, but he has also defined a proof theory for that language in which, he claimed, known properties that are satisfied by BDI systems using BDI Logics [11] could also be proved. Further, he claimed that there is an one-to-one correspondence between his interpreter and the proof system. In this way, he proposed what could be considered as the first viable approach to bridging the ever so quoted gap between BDI theory and practice.

Also, AgentSpeak(L) has many similarities with traditional logic programming, which is another characteristic that would favour its becoming a popular language: it should prove quite intuitive for those familiar with logic programming. Besides, it has a neat notation, thus providing quite elegant specifications of BDI agents.

Further formalisation of the AgentSpeak(L) abstract interpreter and missing details were given in [3] using the Z formal specification language. Most of the elements in that formalisation had already appeared in [2]; this highlights the fact that AgentSpeak(L) is strongly based on the experience with dMARS [5]. In [9], operational semantics to AgentSpeak(L) was given following Plotkin's structural approach; this is a more familiar notation than Z for giving semantics to programming languages.

Despite all these advantages of AgentSpeak(L), until recently there was no implemented interpreter for it. The very first working interpreter for AgentSpeak(L) was presented in [7], which we called SIM_Speak. This was a prototype interpreter based on Sloman's SIM_AGENT toolkit [12]. For this project, we have developed an efficient interpreter in C++, and we have extended the language so as to improve it and to integrate with DTC (as we shall see in Section 3).

We next cover only the basics of the syntax and informal semantics of AgentSpeak(L) first given in [10] (the few formal definitions we found useful to include in this section are actually taken from that paper). This is important for the understanding of the remainder of the paper, in particular the examples from the case study given in Section 4. For detailed formalisation of the language, refer to [3, 9].

An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. The definitions below introduce the necessary notions for the specifications of such sets. Those familiar with Prolog, when reading actual examples of AgentSpeak(L) programs will notice many similarities, including the convention of using uppercase initials for variable identifiers, and the issues related to predicates and unification.

A *belief atom* is simply a predicate in the usual notation, and belief atoms or their negations are *belief literals*. The initial set of beliefs is in fact just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are predicates (as for beliefs) prefixed with the '`!`' operator, while test goals are prefixed with the '`?`'operator. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, they start off the execution of subplans.) Test goals state that agents wants to test whether the associated predicate is a true belief (i.e., whether it can be unified with that agent's base beliefs).

Next, the notion of *triggering event* is introduced. It is a very important concept in this language, as triggering events define which events may start off the execution of plans (the idea of *event*, both internal and external, will be made clearer below). There are two types of triggering events: those related to the *addition* ('`+`') and those related to the *deletion* ('`-`') of mental attitudes (beliefs or goals, in fact).

Clearly, from the usual model of agents (see, e.g., the diagram in [15, page 41]), in regard to their acting on a environment, one sees that plans need to refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as usual predicates, only there are special predicate symbols used for that purpose.

The actual syntax of AgentSpeak(L) programs can be reasonably gathered from the definition of plans below. Recall that the designer of an agent using AgentSpeak(L) does so by specifying a set of beliefs and a set of plans only. An AgentSpeak(L) plan has a head which is formed of a triggering event (denoting the purpose for that plan), and a conjunction of belief literals forming a context that needs to be satisfied if the plan is to be executed (the context must be a logical consequence of that agent's current set of beliefs). A plan has also a body, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test).

DEFINITION 1 (PLAN). *If $e$ is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then* $e$ `:` $b_1$ `&` ... `&` $b_m$ `<-` $h_1$ `;` ... `;` $h_n$ `.` is a *plan*. The expression to the left of the arrow is referred to as the *head* of the plan and the expression to the right of the arrow is referred to as the *body* of the plan. The expression to the right of the colon in the head of a plan is referred to as the *context*. A plan's empty body or context is replaced with the expression "$true$".

We now turn to providing the basics on the interpretation of AgentSpeak(L) programs. Intentions are particular courses of actions to which an agent has committed in order to achieve a particular goal. Each *intention* is a stack of *partially instantiated plans*, i.e., plans where some of the variables have been instantiated.

An *event*, which may start off the execution of plans, can be *external*, when originating from perception of the agent's environment, or *internal*, when generated from the agent's own execution of a plan (e.g., an achievement goal within a plan body is an addition of goal which may be a triggering event). In the latter case, the event is accompanied by the intention which generated it.

The formal definition of an AgentSpeak(L) agent is as follows.

DEFINITION 2 (AGENT). *An* agent *is given by a tuple $\langle E, B, P, I, A, \mathcal{S}_\mathcal{E}, \mathcal{S}_\mathcal{O}, \mathcal{S}_\mathcal{I}\rangle$, where $E$ is a set of events, $B$ is a set of base beliefs, $P$ is a set of plans, $I$ is a set of intentions, and $A$ is a set of actions. The selection function $\mathcal{S}_\mathcal{E}$ selects an event from the set $E$; the selection function $\mathcal{S}_\mathcal{O}$ selects an option or an applicable plan from a set of applicable plans; and $\mathcal{S}_\mathcal{I}$ selects an intention from the set $I$.*

In [7], we have devised a diagram which explains informally the functioning of an interpreter for AgentSpeak(L) (note that this is formalised in [10] and [3]). The pictorial description of such interpreter, given in Figure 1, greatly facilitates the understanding of the interpreter for AgentSpeak(L) proposed by Rao. In the figure, sets (of beliefs, events, plans, and intentions) are represented as rectangles. Diamonds represent selection (of one element from a set). Circles represent some of the processing involved in the interpretation of AgentSpeak(L) programs.

At every interpretation cycle of an agent program, AgentSpeak(L) updates a list of events, which may be generated from perception of the environment, or from the execution of intentions (when subgoals are specified in the body of plan). Note that we have introduced a Belief Revision Function (BRF) in the

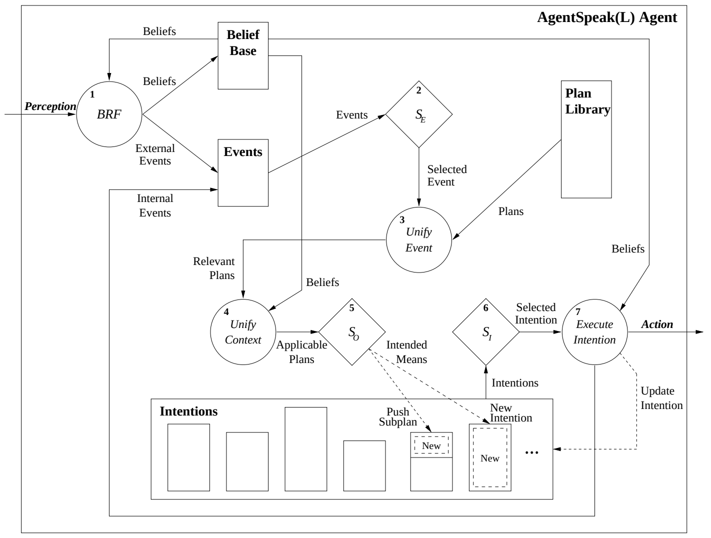


**Figure 1: Interpreting AgentSpeak(L) Programs [7]**

architecture which is implicit in Rao's interpreter (but normally made explicit in the generic BDI architecture [15]). It is assumed that beliefs are updated from perception and whenever there are changes in the agent's beliefs, this implies the insertion of an event in the set of events. We have made this process explicit in the figure, by including the BRF component.

It is important to remember that $\mathcal{S}_\mathcal{E}$, $\mathcal{S}_\mathcal{O}$, and $\mathcal{S}_\mathcal{I}$ (see Definition 2) are part of the definition of an agent. Previous work on AgentSpeak(L) did not elaborate on how users specify such functions, but they are assumed to be agent-specific. After $\mathcal{S}_\mathcal{E}$ has selected an event, AgentSpeak(L) has to unify that event with triggering events in the heads of plans. This generates a set of all *relevant plans*. When unifying the context part of heads of plans in that set with the agent's base beliefs, AgentSpeak(L) determines a set of *applicable plans* (plans that can actually be used for handling the chosen event). Then $\mathcal{S}_\mathcal{O}$ chooses a single applicable plan from that set, and either pushes that plan on the top of an existing intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external, i.e., generated from perception of the environment). Each of an agent's intentions is, therefore, a stack of partially instantiated plans.

All that remains to be done at this stage is to select a single intention to be executed in that cycle. Note that each external event for which there is an applicable plan generates an independent stack of partially instantiated plans within the set of intentions. The $\mathcal{S}_\mathcal{I}$ function selects one of the agent's intention (i.e., one of the independent stacks of plans within the set of intentions). On the the top of that intention there is a plan, and the formula in the beginning of its body is taken for execution. This implies that either a basic action is performed by the agent on its environment, an internal event is generated (in case the subgoal is an achievement goal), or a test goal is performed (which means that the set of beliefs need to be consulted). If the intention is a basic action or a test goal, the set of intention needs to be updated. In the case of test goals, further variable instantiation will occur in the partially instantiated plan which contained that test goal (and the test goal itself is removed from the intention from which it was taken). In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention. When a removed formula marks the end of the body of a subplan, the subgoal that generated it (which therefore stays in the beginning of the body of the plan immediately below it in the stack) is also removed from the intention, or the whole intention is removed from the set if the initial plan (i.e., the plan triggered by an external event) is the one that finished execution. This ends a cycle of execution, and AgentSpeak(L) starts all over again, checking the state of the environment after agents have acted on it, generating events, and so forth.

## 2.2 TÆMS and DTC

The approach to multi-agent systems surveyed in [6] is based on the TÆMS (Task Analysis, Environment Modeling, and Simulation) domain-independent framework [1] to represent formally the coordination aspects of problems. The TÆMS framework deals with worth-oriented environments where a goal is not either fully achievable or not at all, but rather has a degree of achievement associated with it. Various task structures can be active at a time, representing several objectives all of which must be achieved to some extent. The agent's view of the task structure may change over time due to uncertainty, or due to a dynamically changing environment. TÆMS also provides ways to model scenarios where tasks have deadlines and particular kinds of results must be achieved. In that case, those tasks' quality is said to have been accrued.

The central representation in TÆMS is that of the local and non-local sets of activities called task structures, in which several important pieces of information are captured. These include: (a) the top-level goals/objectives/abstract-tasks that an agent intends to achieve; (b) one or more of the possible ways that they could be achieved, expressed as an abstraction hierarchy whose leafs are basic action instantiations, called methods; (c) a precise, quantitative definition of the degree of achievement in terms of measurable characteristics such as solution *quality*, *cost*, and *duration*; (d) task relationships that indicate how basic actions or abstract task achievement affect task characteristics elsewhere in the task structure. The quality of a task group depends on what and when its subtasks and their methods are executed. For example, quality can be accrued by functions such as $q\_min()$ that indicates that all subtasks need be accomplished, and $q\_seq\_min()$ that indicates that all subtasks need be accomplished in the exact order they have been specified. Besides the local effects of the execution of methods on the quality and duration of their supertasks, there exist non-local effects (NLE) such as *enables*, *facilitates*, *hinders*, and so on. A task $T$ may enable a method $M$ in the sense that the quality of $M$ cannot be accrued until $T$ is completed; i.e., the earliest start time of $M$ is the finish time of $T$. Therefore, enables is a hard relationship, which means it must be observed in all cases. When a task $T_1$ facilitates another task $T_2$, the duration and/or quality of $T_2$ is affected, but may not necessarily be observed since facilitates is a soft relationship.

For the work presented in this paper, we have been using TÆMS as a representation language, and also the Design-To-Criteria Scheduler (DTC) [14]. It uses a domain-independent, real-time, flexible computation approach to task scheduling. DTC efficiently reasons about the quality, cost, and duration of interrelated methods, and constructs a set of satisfying schedules for achieving high-level goals. DTC is also part of the recently proposed Soft Real-Time Agent Architecture (see, e.g., [13]). At the moment, we use its DTC module alone, although we plan to use that more efficient architecture in future work.

## 3. INCORPORATING THE DTC SCHEDULER INTO AgentSpeak(XL)

This section outlines the major changes that we have proposed to the AgentSpeak(L) language and its abstract interpreter, both for adding general programming features, and in particular for accommodating DTC as an on-the-fly generator of schedules for intended means (i.e., plans), so that an efficient intention selection function can be used as part of the interpreter. We call this extended version of the language and its interpreter AgentSpeak(XL).

### 3.1 Language Extensions

We have been working on some extensions to AgentSpeak(L) to improve various deficiencies of the language for practical programming. These include, for example, the absence of basic arithmetic and relational operators. Other deficiencies are more significant at a conceptual level of multi-agent systems. One serious disadvantage of AgentSpeak(L) in comparison to other abstract agent oriented programming languages, e.g. 3APL [4], is that it does not provide ways for dealing with plan failure. In fact, Rao pointed out to AgentSpeak(L) events for goal deletion, syntactically represented as $-!g(\mathbf{t})$ and $-?g(\mathbf{t})$, which supposedly were intended for dealing with plan failures, but he did not include that in the semantics of the language.

We have defined a precise mechanism for allowing programmers to use such events in order to handle plan failures. Another significant improvement that we have made is in respect to agent communication. We have devised the means by which AgentSpeak(XL) agents can communicate using a language in the style of KQML [8], and we have defined also the changes to the abstract interpreter that have to be made for agents' mental state to reflect the communications in which they engage. We have also devised a unification algorithm which, unlike what has been suggested in [10], allows the AgentSpeak(XL) programmer to use uninstantiated variables within negated belief atoms in the context part of plans. Finally, we plan some modifications to the handling of events too. We emphasise that, although these extensions are well defined, the formal semantics of these extensions is not as yet fully specified, but under work. We do not give here even informal accounts of these extensions we have defined, as the main focus of this paper is the integration with DTC.

For the integration with DTC, we need to specify labels which unequivocally identify every single plan in the plan library. A plan's label $l$ is separated from the rest of the plan's syntactic structure by a "`->`" symbol. This was the first syntactic change we introduced in AgentSpeak(XL). A plan is now defined as "$l$ `->` $e$ `:` $b_1$ `&` ... `&` $b_m$ `<-` $h_1$ `;` ... `;` $h_n$ `.`" (cf. Definition 1).

In order to allow AgentSpeak(XL) programmers to use basic constructs of any programming language, such as basic arithmetic and relational operators, we have extended the language with a feature we call *internal action*. This allows for the access to user-defined, extensible libraries of procedures, which unlike agent's basic actions (here also referred to as *external actions*), do not affect the environment shared by all agents in a society. The fact that they do not affect the environment is an essential part in the semantics of internal actions: this means that they can be instantaneously executed (as they cause no effects in the environment, and therefore in perception). Because these actions are executed instantaneously, unlike external actions which require the interpreter to proceed to a next cycle (waiting for the action to be performed by the environment and then providing the agent with new perception), it means that we can effectively use them in the context as well as in the body of plans. Recall that the context part of a plan has to be fully evaluated when the interpreter is checking for applicable plans for a specific event (this cannot wait for another interpretation cycle). Also, the possibility of using internal actions in the context of plans is quite important, as programmers may quite often need to have access to those library procedures for actually deciding on whether a plan is applicable or not. For example, an internal action which implements relational operators may need to be used in the context part of the plan to make sure that it will not even be considered as applicable (executing that action in the body of the plan would first allow the plan to be applicable, possibly chosen as intended means, and only later failing; in that case it would be necessary to use the plan failure operators).

Syntactically, internal actions have a '.' in their names, which is used to separate the name of the library from the name of the action (as with C++ classes and methods). This has two advantages: (i) the interpreter can differentiate, among the formulæ in the context of a plan, which are the predicates that need to be checked (against the agent's set of beliefs) for logical consequence, from the ones that are internal actions (recall that their semantics assures that they can be performed in that same interpretation cycle), or differentiate, in the body of plans, internal from basic actions (which are dealt with by the environment and whose effects are only perceived in the next interpretation cycle); and (ii) programmers can organise newly defined actions in various libraries.

A *standard library* is provided with AgentSpeak(XL), which defines useful operators (e.g., relational and arithmetic ones). This is the only "nameless" library, so one can use $.gte(X, Y)$ to access the $gte$ internal action defined in the standard library. Using something like "... `&` $(X >= Y)$ `&` ..." in a plan context is automatically translated into "... `&` $.gte(X, Y)$ `&` ...", which in turn accesses to the standard library.

This definition of internal actions has also been quite handy in the aimed integration with DTC. Besides the standard library, another one, called the task structure library, is available in AgentSpeak(XL). This is the extension of main interest in this paper, as it allows the use of DTC for intention selection. This library is presented in Section 3.3, as we first have to define the changes to the AgentSpeak(L) interpreter that were necessary for integrating with DTC. The other extensions briefly mentioned in this section are not presented in this paper.

### 3.2 Extensions to the Interpreter

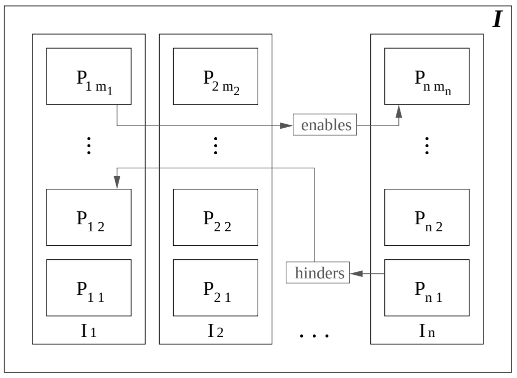
In order to have an efficient intention selection function ($S_{\mathcal{I}}$ in Definition 2 and Figure 1; see Section 2.1), the idea that we are presently pursuing is the following. The DTC scheduler (briefly described in Section 2.2) produces, for a given TÆMS task structure, alternative sequences in which an agent should execute the methods in that task structure so as to best satisfy the criteria (quality, duration, and cost) and deadlines specified for them in the task structure of interrelated methods. Therefore, if we create a TÆMS task structure where the method labels are in fact the labels that uniquely identify the instances of plans that are currently in the set of intentions, and the programmer can set specific values for the scheduling criteria of each plan, as well as define the (TÆMS-like) relations of each plan to the others, then applying DTC to this task structure generates an order in which the plans that are candidate for intention selection would be best chosen for execution.

One important alteration in the data structures used to store the set of intentions was necessary for the integration with DTC. We have to store the current scheduling criteria and relations for each intended means[1] together with it in the set of intentions, so as to be able to generate the TÆMS task structure representing the particu-

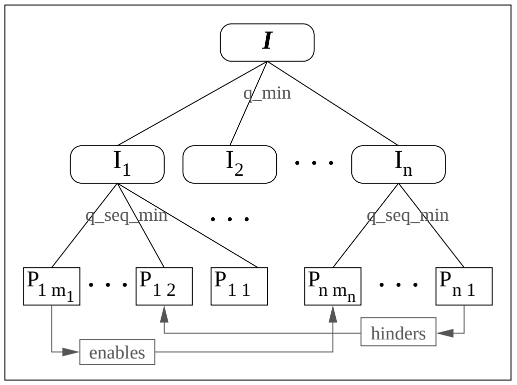
[1]Recall that an intended means is a partially instantiated plan currently in the agent's set of intentions.

lar state of the set of intentions. Note that the criteria and relations can change during the execution of a plan (as seen in the next section); that is why they have to be stored in the set of intentions.

We now further define the translation of a set of intentions into a TÆMS task structure. The root task will have as subtasks each of the agent's intention (i.e., the subtasks represent the stacks of instantiated plans in the set of intentions). This task has quality accumulation function $q\_min()$, as all intentions have to be executed eventually (but in any order). Each intention in turn has as subtasks the plans in its stack of plans, with $q\_seq\_min()$ for quality accumulation function (as the order is important here). A TÆMS method label will in fact be a reference (through plan labels) to the intended means. This provides a simple algorithm for translating an agent's current set of intentions into a TÆMS task structure; this is clearly illustrated in Figure 2, which is self-explanatory.



(a) Set of Intentions



(b) Corresponding TÆMS Task Structure

**Figure 2: Converting a Set of Intentions into a TÆMS Task Structure**

Given that we can translate the set of intentions into a TÆMS task structure (as seen in the figure), and we can run DTC on it, it is easy to create an efficient intention selection function for the AgentSpeak(XL) interpreter: it suffices to read the schedules of plan labels produced by DTC. All the intention selection function has to do is select the first formula (e.g., an action or a subgoal) in the body of the intended means whose label is in the beginning of a schedule provided by DTC. When the formula being executed marks the end of a plan, that plan's label is also removed from the schedule. In the example in Figure 2, all schedules would have intention $\mathrm{I}_1$ being chosen for execution before $\mathrm{I}_\mathrm{n}$, because $\mathrm{P}_{1\ \mathrm{m}_1}$ enables $\mathrm{P}_{\mathrm{n}\ \mathrm{m}_\mathrm{n}}$.

## 3.3 The Task-Structure Library

In this initial version of the Task Structure library of internal actions, which is accessed by the $ts$ library name, we do not consider the possibility of a method having more than one *outcome*, and we define specific values rather than *distributions* for each of the DTC scheduling criteria (i.e., we consider the *density* to be the maximum one for each of these values). Refer to [14] for the precise definition of these TÆMS-related terms.

Each of the methods which represent intended means (plan instances on top of intention stacks) will have a single outcome, which can be associated to whether the plan finished successfully or not. Therefore, when the parameters for the scheduling criteria of that intended means are set (in the plan itself), they can only refer to that single outcome. In order to inform the AgentSpeak(XL) interpreter of the specific values for the three DTC scheduling criteria, we use the internal action

$$ts.criteria(quality\_value, duration\_value, cost\_value)$$

(defined within the $ts$ library), or the programmer can use specific actions to set one particular criterion: $ts.quality(value)$, $ts.duration(value)$, and $ts.cost(value)$. The $ts.deadline(value)$ action is used to set deadlines for plans. (Note that the parameters to those internal actions can be variables if they have been previously instantiated within the plan.) The first version of the AgentSpeak(XL) $ts$ library shall not make use of TÆMS quality accumulation functions other than $q\_min$ and $q\_seq\_min$.

The following internal actions are used for specifying the TÆMS *enables*, *facilitates*, and *hinders* relations between the plan that executes the action and those given as parameter: $ts.enables(p_1, \ldots, p_n)$, $ts.facilitates(p_1, \ldots, p_n)$, and $ts.hinders(p_1, \ldots, p_n)$, respectively. We are investigating whether other TÆMS relations should be considered. Also, one can annotate those relations at the end of the code by prefixing the internal action with "$p$ ->" to specify that it is for the plan whose label is $p$ that the relation holds.

It is interesting to note that, given that these criteria and relations can be specified not only in the context but also in the body of plans, they can be changed dynamically by AgentSpeak(XL) programmers (as we shall see in an example, in the next section). This is possible because the $ts$ internal actions change the plan schedule criteria and plan relations that are associated with that intended means in the set of intentions.

There are two ways in which the TÆMS task structure generated from the agent's set of intentions can change: either when an intended means executes actions from the $ts$ library which change the criteria and relations among plans, or when a new intended means is inserted in the set of intentions, which means that there is a new TÆMS method to be considered for scheduling. Both of them trigger the updating of the TÆMS task structure corresponding to the current set of intentions and the execution of DTC on that task structure. This is done in order to get a new schedule of intended means labels to be used as an intention selection function.

## 4. A CASE STUDY: THE FACTORY PLANT ROBOT

The scenario we shall use to demonstrate the advantages of AgentSpeak(XL) is as follows. A factory produces certain frozen items stored in *boxes* which a robot has to pack with a special wrapping that prevents decaying of the goods. In the *production* end of the factory, each item is produced at a certain *temperature*, which can be critical before the items are packed: when attempting to *pack* an item whose temperature has raised above those critical levels, that basic action fails (i.e., the robot plan for doing so fails too) and the item is thrown away (thus causing losses for the factory). Also, on every packed item, the robot needs to inform the details of the item (its identification number and its production temperature) to an agent that manages the access to the factory's database (referred to as `db` in the AgentSpeak(L) code below). We concentrate in the reasoning of a type of *robot* in the factory plant which is in charge of packing the items and, when there remains no unpacked items at the production end, the robot has to carry these *processed* items to the storage end of the factory where the *freezers* are located. We assume that there is only one of these robots in the factory plant (to avoid coordination problems, which we do not address in this paper), as well as we assume that the robot can carry any number of items at a time. That is, the robot is always capable of carrying whatever items it has processed since it last went to the freezers.

There is one more task of which the robot has to take care. The freezers' thermostat *settings* need to be changed from time to time according to the specific type of products that have been stored in the freezers. The robot always knows what that setting should be, based on the types of items it has been processing. In the code below, there is a simple plan for *communicating* such temperature settings to the agent that controls the freezers, which includes the transmission of the robot's *identification* and a *passcode* for the freezer agent to grant access to freezer settings. Besides requesting an identification and passcode, as a further security measure, the designers of the factory's information systems have recently decided that the freezer managing agents will, at certain intervals of time, request the robot to confirm the temperature setting it requires for the items it is processing (which means that it has to identify itself and provide the given passcode). However, when changing the settings of a particular freezer, no other freezer settings can be changed by the robot.

This added security measure (of frequent robot identification requests by the freezer agents) would prevent sabotage to the factory production line, but it requires the robot to give priority to the temperature setting plan over its other tasks. If the robot takes too long to finish a temperature setting sequence, the freezer agent that has issued that identification request to the robot will sound the alarm to alert the security system installed in the factory plant. Time is particularly critical between the actions that communicate the parts of the identification and *temperature setting sequence*.

We intend to show that, differently from AgentSpeak(L), with AgentSpeak(XL): i) we can easily improve the performance of the robot for its particular tasks (in regards to avoiding items to be thrown away because they reached critical temperature levels); ii) we can indeed give actual priority to the temperature setting sequence (thus avoiding the alarm being sound); and above all iii) we do so by reusing what seems to be a reasonable library of plans for these tasks with minor changes.

Below, we give straightforward plans for dealing with the various tasks that the robot has to perform, without considering any of the restrictions that there exists in terms of priorities and in the synchronisation between them for this particular scenario. That is, in the simple version below we do not handle the fact that certain plans cannot be executed before others have finished, that all current intentions have to be interrupted when plans with higher priorities also become intended, and so on. In fact, as we shall comment later on, it is quite difficult to do that in AgentSpeak(L). Before we start discussing the plans, it is worth presenting the initial belief base[2] that would be required in the running version of the robot program (both in AgentSpeak(L) and AgentSpeak(XL)).

[2]Recall that AgentSpeak(L) programs are given by a set of initial beliefs and a set of plans.

```
id(robot1).
passcode(initcode).
settings(265).        // 265K = -8°C
location(production).
```

Note that the temperature is expressed in Kelvin so as to simplify the use of a box's temperature in determining the appropriate priority it should be given. We next show the first plan, labelled `b1`, which takes care of packing boxes and recording the relevant information. Recall that these plans express the tasks to be done by the robot quite generally, without any concern for prioritising anyone of them once they become intended means.

```
b1 -> +box(Id,Temp) : location(production) <-
   pack(Id);
   communicate(db,processed(Id,Temp));
   +processed(Id,Temp).
```

All items produced while the robot was away storing previously processed boxes will only be perceived (resulting in the addition of a `box(Id,Temp)` belief) by the robot when it returns to the production end of the factory plant. Only then will all those boxes be processed in parallel (that is, as separate focuses of attention in the robot's set of intentions).

Recall that these boxes' temperatures may be critical. In fact if an item's temperature has raised above a certain threshold when the robot attempts to `pack` it, the basic action fails (it means that the item is thrown away). In consequence, the whole plan fails too (thus preventing the robot from communicating a processed item to the database, etc.). Giving priority to boxes with higher temperatures is easily dealt with in AgentSpeak(XL), while dealing with that in AgentSpeak(L) is quite tedious, unless the added feature of internal actions is used.

```
b2 -> -box(Id,Temp) : not box(AnyId,AnyTemp) <-
   pickAllPacks;
   +destination(freezers);
   moveTowards(freezers);
   !calculateIdealTemp;    // this plan is not expounded
   !deleteAllProc.
```

The perception of a box persists until it is packed; accordingly, an agent's `box(Id,temp)` belief is deleted when it is packed. The last box to be packed triggers the robot to pick all packed boxes, take them to the freezers, store them, and then return to the production end. In parallel with that moving, the robot also calculates the ideal freezer temperature for the processed items (these parallel activities are possible as they are separate focuses of attention in the set of intentions, the one for moving being determined by the external event of having its location changed). Then the robot updates its belief base by removing all beliefs about recently processed items (the ones that are going to be, or have just been, stored in the freezers).

Note that events of type `-box(Id,Temp)`, associated with all boxes that were packed when there remained other boxes ready for packing, will not have applicable plans, and therefore will be discarded. Also, the plan for calculating the ideal temperature for the freezers, considering the particular types of items being processed by the robot, will not be shown here. All we need to know about that plan is that it changes the `settings(T)` belief of the robot, by consulting the types of items it has recently processed.

Plans `d1` and `d2` below delete all beliefs about processed items (when they have already been considered by the plan that calculates the best temperature setting for the freezers). Note also that these deletions must finish before the robot starts again to pack boxes once it has come back to the production end. We shall see in the next section how to assure that happens, both in AgentSpeak(L) and in AgentSpeak(XL). In the plans below we assume that the applicable plan selection function chooses the topmost applicable plan when there is more than one.

```
d1 -> +!deleteAllProc : processed(Id,Temp) <-
   -processed(Id,Temp);
   !deleteAllProc.
d2 -> +!deleteAllProc : true <- true.
```

Plans `l1` to `l3` actually move the robot to either end of the factory plant. First we need to check whether it has arrived to its present destination. If it has arrived to the freezers, all the robot has to do is store all packs and then move back to the production end of the factory plan. It does so by changing its belief on its `destination` and then just moving to one adjacent cell towards that direction, which is done with the basic action `moveTowards`. Plan `l3` will then keep the robot moving (as its location is changed through perception) until it gets to either destination.

```
l1 -> +location(freezers) : true <-
   storeAllPacks;
   -destination(freezers);
   +destination(production);
   moveTowards(production).
```

There is not much to do when the robot is back. The boxes produced while it was away (if any) will all be perceived now by the robot, so the events about box production will keep the robot busy. Boxes produced while the robot is at the production end are immediately perceived.

```
l2 -> +location(production) : true <-
   -destination(production).
```

The next plan keeps the robot moving towards its present destination, either *production* or *freezers* according to its present `destination` belief, until the robot eventually arrives there. It relies on the perception of location being changed. If the location is changed, a belief is changed through perception, generating a (triggering) event. Again we assume that the applicable plan selection function chooses the topmost applicable plan (so `l3` is not selected if `l1` or `l2` apply).

```
l3 -> +location(X) : destination(D) <-
   moveTowards(D).
```

The plan below sends the temperature setting sequence to the freezer that has requested it. Recall that if more than one freezer makes such request, only one can be handled at a time, and it should be given priority over all other focuses of attention of the robot. Also the three `communicate` actions should be executed in three consecutive reasoning cycles: if the robot takes too long to reply to these requests, or if it delays too much between communicated parts of the sequence, the alarm will be sound. We shall see how to guarantee these requirements both in AgentSpeak(L) and AgentSpeak(XL) in the next section.

```
f1 -> +freezer(F) : true <-
   ?id(RobotId);
   ?passcode(PassCode);
   ?settings(Temperature);
   communicate(F,RobotId);
   communicate(F,PassCode);
   communicate(F,Temperature).
```

The next section mentions what changes are necessary to have these plans working for the given scenario both in AgentSpeak(L) and in AgentSpeak(XL). It also presents the results of the simulations using both interpreters.

## 5. RESULTS: COMPARING AgentSpeak(L) AND AgentSpeak(XL)

In order for the plans shown in the previous section to actually work in an AgentSpeak(L) interpreter that uses a generic intention selection function (as the one we presented in [7]), a large number of changes have to be made to that code. Rao assumed, when he defined AgentSpeak(L) [10], that users would provide the three selection functions (see Definition 2) needed by the interpreter (i.e., they were assumed to be agent-specific). However, neither he nor d'Inverno and Luck [3] provided the means for the specification of such intention selection functions. Their implementation in standard programming languages may not be straightforward, posing serious hindrances to the use of AgentSpeak(L). The simple and efficient specification of dependencies between plans allowed in AgentSpeak(XL), with its automatic generation of intention selection functions by DTC, is one of its greatest advantages over the original AgentSpeak(L) programming language.

Using a generic intention selection function (i.e., considering that an application-specific, user-defined, intention selection function is not available), the following changes would have to be made to the plans shown in the previous section for them to work properly in such AgentSpeak(L) interpreter. The first thing we have to worry about is giving the necessary priority to the temperature setting sequence. In fact, full priority is not possible in AgentSpeak(L). If, for example, the freezer agent required the very next robot action to be the first step in the identification and temperature setting plan, and that all steps were to be done with no intervals (in terms of interpretation cycles), that would not be possible at all in AgentSpeak(L). In AgentSpeak(XL), on the other hand, that is quite easily achieved, as we shall see later.

The best we can do in AgentSpeak(L) is to divide the plans above in as many separate plans as possible and in the context of all of them to check whether a freezer agent has requested a temperature setting sequence (by checking whether a particular belief is present in the belief base). If that is the case, the internal event requesting the pushing of a subplan would fail (by the lack of an applicable plan), as all plans are actually only applicable if there is no request from the freezers, thus causing the intended means that generated the internal event to be removed. So there must be alternative plans for each event which, in case there are freezer requests, they record in the agent's belief base what the robot was doing when the interruption from the freezer happened. However, it is important to note that, in case there are too many independent intention stacks in the set of intentions, it may take too long for the priority to be given to the freezer request (i.e., until other intended means on top of intention stacks are eliminated as explained above) and so the alarm may end up being sound by the freezer agent. There is a similar mechanism for assuring that plan `b2` finishes before any instance of `b1` is allowed to start running. Giving priority to higher temperature items was much facilitated by the use of a specific internal action (a construct that was not available in the original AgentSpeak(L) language).

We do not show here the complete code for either implementations of the robot's reasoning (in AgentSpeak(L) and AgentSpeak(XL)). However, by the description above, it is not difficult to see that the AgentSpeak(L) code is not elegant at all. The resulting code is extremely clumsy because of the use of many belief addition, deletion, and checking (for controlling intention selection). It is thus a type of code that is very difficult to implement and maintain. In fact, with a generic intention selection function, the AgentSpeak(L) interpreter does not allow any complete solution to the problem, or even a more elegant implementation. Also, these many extra handling of beliefs do increase the number of reasoning cycles required by the robot, as we see below.

We now describe how an intention selection function generated by DTC solves the problem of giving full priority to a freezer request, once dealing with it becomes an intended means; the problem of prioritising items with higher temperatures; and the problem of assuring that all beliefs about processed items have been removed before the robot starts packing again. (This last requirement is to avoid deleting information on processed items that have not yet been taken into consideration by the plan that calculates the freezers' temperature setting).

All that is required is the inclusion of a few TÆMS-like relationships to the plans, and setting some deadlines for associating instances of the plan for packing items with the difference between a maximum value and the temperature of the items themselves. This way, the higher the temperature of the item, the earlier the plan's associated deadline, hence the sooner it will be packed, preventing items from being thrown away, whenever possible. In order to have a working AgentSpeak(XL) solution to the problem (including the freezer request priority), we start from the general plans (given in the previous section) and simply insert the commands described below.

In the context part of plan `b1`, we include **`& D = 1000-Temp & ts.deadline(D)`**, and then we insert **`ts.deadline(1000);`** right after the basic action `pack(Id);` in the body of that plan. That is, to the original plan, we only need to add the setting of the deadline of the plan, which is earlier for higher temperature items, but only until that item is actually packed. Note how we set, during the execution of the plan itself, the deadline back to the maximum value (we have used 1000 for that) once the item is packed. This assures a higher priority to that plan instance over the instances for other boxes, according to the item's temperature, only while that item is not packed, thus allowing other instances of the plan, handling other unpacked boxes, to have priority of execution.

At the end of the original plans, we annotate some TÆMS-like relationships among them with the following commands:

**`b1 -> ts.enables(b2).`** assures that `b2` only starts to run when all boxes have been packed and processed (it is then time to take the items to the freezers);

**`b1 -> ts.hinders(d1,d2).`** assures that `d1` and `d2` finish before `b1` starts to run (to prevent, when the robot returns to the production end, the deletion of information on recently processed items);

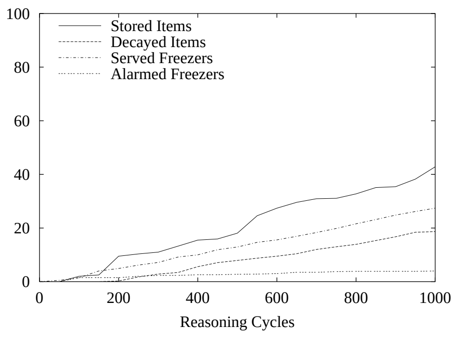
**`f1 -> ts.enables(f1,b1,b2,d1,d2,l1,l2,l3).`** gives priority to attending freezer requests (plan `f1`) over all other goals of the agent.

Priority is given immediately to the first instance of a freezer request plan in the set of intentions. If two freezers make requests at the same time, the second one will be given priority right after the first request is attended. In AgentSpeak(L), the second may need to wait for a priority (until other intentions get stuck), in the same way it happens for a first instance of the plan in the set of intentions.

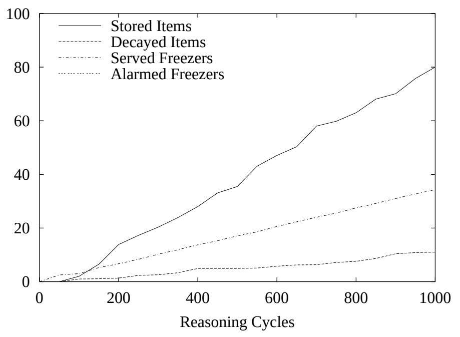
If a plan has a TÆMS relation to itself (as is the case with `f1`), when it is inserted in the set of intentions, we only include a relation of that type from the instances that are already intended means to the new instance being inserted. Therefore, an *enables* relationship from `f1` to itself solves the problem of allowing only one of the freezers to have its temperature set at a time. The relationship to all other types of plans guarantees the priority to `f1` over all of them.

As seen above, only a few changes to the original plans are needed in order to get the automatic set up of an efficient intention selection function by using DTC. Also, annotating such plan relationships at the end of the source code, rather than in the plan context or body (which is also possible), makes it easier to reuse and maintain plans, as the plans themselves need not be changed. Furthermore, the AgentSpeak(XL) version not only works for all required restrictions on the original plans, but it is also quite efficient, as seen below by the results obtained for the factory plant scenario simulation.

We have run simulations of the factory plant scenario where we have set two parameters: the maximum number of boxes that could have been produced every time the robot arrived at the production end (either 10, 20, 30, or 40), and the average number of interpretation cycles between requests for temperature setting issued by each freezer (either 60, 80, or 100). This gives a total of 12 different configurations. The factory floor is divided into cells, and the robot can move from one of these to an adjacent one each time it selects the basic action for moving (thus taking one interpretation cycle). The distance between the production end and the freezers is 10 cells, and there are 3 freezers in the factory plant. The results obtained for both interpreters can be seen in Figure 3, where we show the average results over the twelve different configurations we have used. We do not show or comment on the individual ones for conciseness, but AgentSpeak(XL) has a consistently better performance.



(a) AgentSpeak(L)



(b) AgentSpeak(XL)

**Figure 3: Average Results over Various Configurations**

From the figure, one can see clearly that the AgentSpeak(XL) robot is able to store almost double the number of items stored by the AgentSpeak(L) robot, whereas the number of decayed items is almost halved. The AgentSpeak(XL) robot has allowed no freezer alarms to be sounded, whilst the AgentSpeak(L) robot has failed to reply in time to around $10\%$ of freezer requests.

## 6. CONCLUSION

We have integrated the DTC scheduler into an AgentSpeak(L)-like programming language that we have called AgentSpeak(XL). Some new constructs were added to that language, for its general improvement, and to accommodate the DTC-based generation of intention selection functions. The extended interpreter allows, among other things, the automatic conversion of the agent's set of intentions into a TÆMS task structure so that its DTC schedules define efficient intention selection functions. By the specification of plan relationships and quantitative criteria with the added language constructs, which is done in a high level and easy way, we allow programmers to have control over intention selection, a job that was particularly difficult to do with AgentSpeak(L), as seen in our case study on the factory plant robot.

Our more ambitious goal is to further integrate, in a two-level agent architecture, the cognitive and utilitarian approaches to multi-agent systems. That should be done by allowing AgentSpeak(XL) agents to use GPGP2 [6] coordination mechanisms (when they autonomously deliberate to do so). Meanwhile, our very next step is to use other components of the Soft Real-Time Agent Architecture [13] rather than DTC alone. That should increase efficiency of the interpreter in regards to the automatic generation of intention selection functions, considering that it would avoid the need to reschedule whenever the set of intentions is changed.

Further evaluation of the results we obtained from the case study presented here should help us assess whether our decisions to schedule whole plans rather than basic (external) actions was a good one. We also have to investigate the use of DTC for implementing the other AgentSpeak(L) selection functions, in particular the event selection function. Undoubtedly, tackling the intention selection problem was more important than the others, considering that there is a natural applicable-plan selection function (namely, using the order in which the plans were given by the programmer, as in Prolog), and we are working on a version of AgentSpeak(XL) that tries to handle as many events as possible in a single interpretation cycle. Theoretical work is also being done on giving formal semantics to our extensions of AgentSpeak(L), based on the semantics given in [9].

Pursuing the ideas we have presented here has been quite illuminating about the integration of the frameworks with which we have been working. It is a significant step towards our more ambitious goal of reconciling utilitarian coordination to cognitive agents.

## Acknowledgements

We are grateful to Rodrigo Machado for many discussions on a former implementation of a prototype AgentSpeak(L) interpreter, to Renata Vieira for comments on the case study scenario, and to Sherief Abdallah for suggestions on the conversion of the intention set to a TÆMS task structure.

## 7. REFERENCES

[1] K. S. Decker and V. R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 2(4):215–234, 1993.

[2] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24–26 July, 1997*, number 1365 in LNAI, pages 155–176. Springer-Verlag, Berlin, 1998.

[3] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):1–27, 1998.

[4] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Control structures of rule-based agent languages. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V—Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), held as part of the Agents' World, Paris, 4–7 July, 1998*, number 1555 in LNAI, pages 381–396, Heidelberg, 1999. Springer-Verlag.

[5] D. Kinny. The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[6] V. R. Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. *Autonomous Agents and Multi-Agent Systems*, 1(1):89–111, 1998.

[7] R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In *Pre-Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, 2001. Proceedings to appear as a volume of LNAI – Intelligent Agents Series.

[8] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, number 1037 in LNAI, pages 347–360, Berlin, 1996. Springer-Verlag.

[9] Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22–25, Toulouse, France*, 2002.

[10] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.

[11] A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.

[12] A. Sloman and R. Poli. SIM_AGENT: A toolkit for exploring agent designs. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, number 1037 in LNAI, pages 392–407, Berlin, 1996. Springer-Verlag.

[13] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing soft real-time agent control. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of The Fifth International Conference on Autonomous Agents (Agents 2001), 28 May – 1 June, Montreal, Canada*, pages 355–362. ACM Press, 2001.

[14] T. Wagner, A. Garvey, and V. Lesser. Criteria-directed heuristic task scheduling. *International Journal of Approximate Processing, Special Issue on Scheduling*, 19(1–2):91–118, 1998.

[15] M. Wooldridge. Intelligent agents. In G. Weiß, editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–77. MIT Press, Cambridge, MA, 1999.