Augmenting Agent Computational Environments with Quantitative Reasoning Modules and Customizable Bridge Rules

Stefania Costantini[1] and Andrea Formisano[2]

[1] DISIM, Università di L'Aquila
[2] DMI, Università di Perugia, GNCS-INdAM

Abstract. There are many examples where large amount of data might be potentially accessible to an agent, but the agent is constrained by the available budget since access to knowledge bases is subject to fees. There are also several activities that an agent might perform on the web where one or more stages imply the payment of fees: for instance, buying resources in a cloud computing context where the objective of the agent is to obtain the best possible configuration of a certain application withing given budget constraints. In this paper we consider the software-engineering problem of how to practically empower agents with the capability to perform such kind of reasoning in a uniform and principled way. To this aim, we enhance the ACE component-based agent architecture by means of a device for practical and computationally affordable quantitative reasoning, whose results actually determine one or more courses of agent's actions, also according to policies/preferences.

1 Introduction

There are many examples where large amount of data might be potentially accessible to an agent, but the agent is constrained by the available budget since access to knowledge bases is subject to fees. There are also several activities that an agent may perform on the web on behalf of an user where one or more stages imply the payment of fees. An important example is that of buying resources in a cloud-computing context, where the objective of the agent is to obtain the best possible configuration for performing certain tasks in the sense of maximizing performance and minimizing costs, that can anyway stay withing given budget constraints. The work [33] identifies the problem that an agent faces when it has limited budget and costly queries to perform. In order to model such situations, the authors propose a special resource-aware modal logic so as to be able to represent and reason about what is possible to do with a certain available budget. The logic can be adapted to reason separately about cost and time limitation, though an integration is envisaged. Interesting as it is, this work constitutes a good starting point but it presents two problems: (i) such kind of modal logic is computationally hard (though this aspect is not discussed in the aforementioned paper) and thus it can hardly constitute the basis for practical tools; (ii) the axiomatic system of [33] allows one to prove that something can or cannot be achieved within a certain cost. However, an agent needs, in general, to become aware of how goals might possibly be achieved, and should be enabled to choose the best course of action according to its own policies/preferences.

In this paper we tackle some issues related to this problem. First, we consider the software-engineering problem of how to practically empower agents with the capability to perform such kind of reasoning in a uniform and principled way. Second, we consider the adoption of a reasoning device that enables an agent, which may have several costly objectives, to establish which are the alternative possibilities within the available budget, and to select, based upon its preferences, the goals to achieve and the resources to spend, and finally to implement its choice.

Concerning the first aspect, we enhance the Agent Computational Environment (ACE) framework [13], which is a software engineering methodology for designing intelligent logical agents in a modular way. Therefore, in this paper we refer to agent-oriented languages and frameworks which are rooted in Computational Logic. Modules composing an agent interact, in ACE, via *bridge rules* in the style of the Multi-Context Systems (MCS) approach [7, 8, 10]. Such rules take the form of conjunctive queries where each conjunct constitutes a sub-query which is posed to a specific module. Thus, the result is obtained by combining partial results obtained from different sources. The enhancements that we propose here for ACE are based upon the flexible agent-tailored modalities for bridge rules application and for knowledge elaboration defined for the DACMACS framework (Data-Aware Commitment-based managed Multi-Agent-Context Systems), which is aimed at designing data-aware multi-agent-context systems [14, 15]. There, bridge rules are proactively triggered upon specific conditions and the obtained knowledge is reactively elaborated via a *management function* which generalizes the analogous MCS concept.

Second, we extend ACEs so as to include modules for specialized forms of reasoning, including quantitative reasoning. For this kind of reasoning we suggest to adopt the RASP framework [17, 19, 16], which is based upon Answer Set Programming (ASP) and hence it is computationally affordable and reasonably efficient. We show the suitability of such approach by discussing a case study, that will constitute the leading example throughout the paper.

A strong innovation that this paper proposes is that, after obtaining from a reasoning module the description of possible courses of actions, bridge rules "patterns" can be specialized and activated so as to put them into action. This feature is made possible by an enhanced flexible ACE semantics.

The resulting framework can be seen as a creative blend of existing technologies, with some relevant formal and practical extensions. Partially specified bridge rules and their dynamic customization and activation is an absolute novelty and constitutes a relevant advance over MCSs versions, applications and extensions: in fact, bridge rules have been so far conceived as predefined, ground and not amenable to any adaptation. Beyond quantitative reasoning, such more general bridge rules may constitute a powerful flexible device in many applications.

The paper is organized as follows. Section 2 presents a case study that will constitute the leading example throughout the paper. In Section 3 we discuss the quantitative reasoning device we suggest to exploit. Sections 4 and 5 present the enhanced ACE framework and illustrate, on the case study, the dynamic customization of bridge rules. Section 6 introduces the extended ACE semantics and for completeness we provide in Section 7 an actual RASP formalization. Concluding remarks are given in Section 8.

2 Specification of the Case Study

In this section we provide the specification of a case study which we will adopt in the rest of the paper for the illustration of the proposed enhancements to the ACE framework. In Section 7 we will present a realistic implementation in a specific existing approach for quantitative reasoning, shortly introduced in the next section.

We consider a student, that will be represented by an agent which can be seen as her "personal assistant agent". Upon completing the secondary school, she wishes to apply for enrollment to an US university. Each application has a cost, and the tuition fee will have to be paid in case of admission and enrollment. The student has an allotted maximum budget for both. Thus the agent, on behalf of the student, has to reason about: (i) the universities to which an application will be sent; (ii) the university where to enroll, in case a choice can be made.

Actually, the proposed case study is seen as a prototype of a wide number of situations where two kinds of quantitative reasoning are required:

1. The cost of knowledge, as in practical terms a student applies in order to know whether she is admitted.
2. Reasoning under budget limits, as a student may send an application only if: (i) she can afford the fees related to the application; (ii) in case of admission, she can then afford the tuition fees.

If a solution is found considering her preferences and her budget, she will then be able to apply and, if admitted, to enroll. In case more than an option is available, a choice is required so as to select the "best" one according to some criteria.

Without any pretension to precision, we consider the steps that a student has to undergo in order to apply for admission:

1. Pass the general SAT test.
2. Pass the specific SAT test for the subject of interest (such as Literature, Mathematics, Chemistry, etc.)
3. In case of foreign students, pass the TOEFL test.
4. Fill the general application on the application website (that we call collegeorg).
5. Send the SAT results to the universities of interest.
6. Complete the application for the universities of interest.

All these steps are subject to the payment of fees, which are fixed (the fee is independent of the university) for steps 1-4 and depend upon the selected university for steps 5-6. In the example we assume that the student has a budget for the application (say 1500 US dollars) and a limit about the tuition fee she is able to pay (say 22000 US dollars per year). However, she has a list of preferred universities, and within such list she would apply only to universities whose ranking is higher than a threshold. Additionally, since she likes basketball, all other things being equal (*ceteris paribus*) she would prefer universities with the best rankings of the basketball team.

3 Resource-based Reasoning

In the case study, the student's personal assistant agent needs the support of some kind of quantitative reasoning module. Such module should in general be able to provide the agent, given one or more objectives, with a description of the different ways of achieving the objectives while staying within a budget. A desirable property of the reasoner would be that of allowing preferences and constraints to be expressed about objectives to achieve and modalities for achieving them. A mandatory requisite is the ability to perform such reasoning in a computationally affordable way.

In knowledge representation and reasoning, forms of quantitative reasoning are possible, for example, in Linear Logics and Description Logics. For Linear Logic in particular, several programming languages and theorem provers based on its principles exist (cf. [16] for a discussion). In this paper we adopt RASP (Resource-based ASP) [17, 19], which has in fact been proven in [18] to be equivalent to an interesting fragment of Linear Logic, specifically, to an empowered Horn fragment allowing for a default negation that Linear Logic does not provide (though still remaining within an NP-complete framework). RASP extends ASP, which is a well-known logic programming paradigm where a program may have several "models", called "answer sets", each one representing a possible interpretation of the situation described by the program (cf., among many, [29]). In particular, RASP explicitly introduces in ASP the notion of *resource*, and supports both formalization and quantitative reasoning on consumption and production of resources. RASP also provides complex preferences about spending resources (and in this it is different from the several approaches to preferences that have been defined for ASP, see e.g., [2, 6, 11, 25] and the references therein). Compared with the "competitors", RASP represents possible different uses of a resource and non-determinism in general by means of different answer sets, rather than exploring the various possibilities via backtracking in a Prolog-like fashion. The RASP inference engine is based upon publicly available ASP solvers [35] that are remarkably well-performing and subject of intensive research and development. After the seminal work of [34] one can mention [26, 28, 32, 1, 31, 22], among the most recent developments. Specifically, RASP execution is based upon a front-end module called *Raspberry* which translates RASP programs (via a non-trivial process, see [19] for the details) into ASP. The resulting program can be executed by common ASP solvers.

As a side note, we observe that the clasp ASP solver allows one to add external functions to ASP programs. This is done by defining deterministic functions in a scripting language such as lua or python. Relying on this possibility, one might envisage a re-implementation of the RASP framework exploiting such feature of this specific ASP solver, instead of performing a translation from RASP into ASP, as done in Raspberry. Another recently proposed extension of ASP is H-ASP [12], where propositional reasoning is combined with external sources of numerical computation. The main aim of H-ASP is to allow users to reason about a dynamical system by simulating its possible evolutions along a discretized timeline. The external computations are used to compute the system transitions and may involve both continuous and discrete numerical variables. The expressive power of the resulting framework directly depends on the kind of numerical tasks one integrates, and the computational complexity can exceed NP. Clearly, thanks to the generality of ACE, one could integrate modules based on H-ASP in the ACE framework, similarly to what done for RASP. However, in the case of RASP we stay within NP and directly rely on common "pure-ASP" engines without the need of integrating (and encoding) further computational services.

We are not aware of other reasoning frameworks that combine logic and quantitative techniques, apart from the one proposed in [33], which however is not implemented and, as mentioned, in its present form can hardly admit a computationally affordable version. So, there is nowadays no competitor approach to RASP in practical logic-based quantitative reasoning and its applications in agent systems.

4 Enhancing the ACE Framework

The ACE framework as defined in [13] considers an agent as composed of:
1) the "main" agent program;
2) a number of Event-Action modules for Complex Event Processing;
3) a number of external contexts the agent can access in order to gather information.

ACE is therefore a highly modular architecture, where the composing modules communicate via *bridge rules* (to be seen below) in the style of Multi-Context Systems (MCSs) [7, 8, 10]. MCSs constitute in fact a particularly interesting approach for modeling information exchange among heterogeneous sources because, within a neat formal definition, it is able to accommodate real heterogeneity of sources by explicitly representing their different representation languages and semantics. The same holds for ACEs, where: external contexts are understood as in MCS, i.e., they can be queried but cannot be accessed in any other way; and where "local" agent's modules (main agent program and event-action modules) can be defined in any agent-oriented computational-logic-based programming language, such as, e.g., DALI, AgentSpeak, GOAL, 3APL, METATEM, KGP, etc. (see [3, 4, 5, 20, 21, 24, 27, 30] and the references therein), or also in other logic formalisms such as, e.g., ASP (see [29] and the references therein).

In the present setting, we augment the framework with a set of *Reasoning Modules*, say $R_1, \ldots, R_q$, $q \geq 0$, that we see as specialized modules which are able to perform specific forms of reasoning by means of the best suitable formalism/technique/device. Among such modules we may have quantitative reasoning modules. Therefore, an (enhanced) Agent Computational Environment $\mathcal{A}$ is now defined as a tuple

$$\left\langle A, M_1, \ldots, M_r, C_1, \ldots, C_s, R_1, \ldots, R_q \right\rangle$$

where module A is the "basic agent", i.e., an agent program written in any agent-oriented language. The "overall" agent is obtained by equipping the basic agent with the following facilities. The M_is are "Event-Action modules", which are special modules aimed at Complex Event Processing, that allow the agent to flexibly interact with a complex changing environment. The R_js are "Reasoning modules", which are specialized in specific reasoning tasks. The C_ks are contexts in the sense of MCSs, i.e., external data/knowledge sources that the agent is able to query about some subject, but upon which it has no further knowledge and no control: this means that the agent is aware of the "role" of contexts in the sense of the kind of knowledge they are able to provide, but is unable in general to provide a description of their behavior/contents or to affect/modify them in any way.

Interaction among ACE's components occurs via *bridge rules*, inspired by those in MCS. They can be seen as Datalog-like queries where however each sub-query can be posed to a different module. In MCS, bridge rules have, in general, the following form:

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathit{not}\, (c_{j+1} : p_{j+1}), \ldots, \mathit{not}\, (c_m : p_m).$$

The meaning is that the rule is *applicable* and s can thus be added to the consequences of a module's knowledge base whenever each atom p_r, $r \leq j$, belongs to the consequences of module c_r (that can be a context or an event-action module, or the basic agent), while instead each atom p_w, $j < w \leq m$, does not belong to the consequences of c_w. Practical run-time bridge-rule applicability will consist in posing query p_i to context c_i. In case for some of the c_is the context is omitted, then the agent is querying its own knowledge base. The part $(c_1 : p_1), \ldots, (c_j : p_j)$ is the *positive body* of the rule, while the remaining part is the *negative body*.

We introduce the following restriction on bridge rules bodies: the basic agent A can query any other module (and, clearly, if it is situated in a MAS context it can communicate with other agents according to some kind of protocol). The M_is and the R_is can query external contexts and the basic agent. Contexts can only query other contexts, i.e., they cannot access agent's knowledge. We also assume (for simplicity and without loss of generality) that bridge-rule heads are unique, i.e., there are never two bridge rules with the same head.

In Managed MCSs the conclusion s, which represents the "bare" result of the application of the bridge rule, becomes $o(s)$ where o is a special operator, whose semantics is provided by a module-specific *management function*. The meaning is that the result computed by a bridge rule is not blindly incorporated into the "target" module knowledge base. Rather, it is filtered, adapted, modified and elaborated by an operator that can possibly perform any elaboration, e.g. evaluation, format conversion, belief revision. To the extreme, the new knowledge item can even be discarded if not deemed to be useful.

In the basic agent we adopt, with suitable adaptations, the special agent-oriented modalities introduced in DACMACS. There, bridge-rule activation and management-function application has been adapted to the specific nature of agent systems. First, while bridge rules in MCSs are conceived to be applied whenever applicable (they can be seen, therefore, as a reactive device), DACMACS provides a proactive application upon specific conditions. Second, the incorporation of bridge rule results via the management function is separated from bridge-rule application. In particular, bridge-rule application is determined by a *trigger rule* of the form

$$Q \textbf{ enables } A(\hat{x})$$

where: Q is a query to agent's internal knowledge-base and $A(\hat{x})$ is the conclusion of one of agent's bridge rules. If query Q (the "trigger") evaluates to true, then the bridge rule is allowed to be applied. A trigger rule is proactive in the sense that the application of a bridge rule is enabled only if and when the agent during its operation concludes Q. The bridge rule will be actually applied according to agent's internal control modalities, and will return its results in $\hat{x}$. The result(s) $\hat{x}$ returned by a bridge rule with head $A(\hat{x})$ will then be exploited via a *bridge-update rule* of the following form (where $\beta(\hat{x})$ specifies the operator, management function and actions to be applied to $\hat{x}$):

$$\textbf{upon } A(\hat{x}) \textbf{ then } \beta(\hat{x})$$

We propose a relevant improvement concerning bridge rules. In particular, in MCSs bridge rules are by definition ground, i.e., they do not contain variables: in [9], it is literally stated that [in their examples] they "*use for readability and succinctness schematic* *bridge rules with variables (upper case letters and '_' [the 'anonymous' variable]) which range over associated sets of constants; they stand for all respective instances (obtainable by value substitution)*" where however such "placeholder" variables occur only in the p_is while instead the c_is (contexts' names) are constants. This is a serious expressive limitation, that we have tackled in related work. In fact, we admit variables in both the p_is in bridge-rule bodies and in the head s, to be instantiated at run-time by the queried contexts. We also admit contexts in the body to be selected from a directory according to their *role*. Here, we propose a further relevant enhancement: we allow contexts occurring in the body of the bridge rules of the main agent A to be instantiated via results returned by ACE's other modules. Such bridge rules will have this form:

$$s \leftarrow (\mathcal{C}_1 : p_1), \ldots, (\mathcal{C}_j : p_j), \mathit{not}\,(\mathcal{C}_{j+1} : p_{j+1}), \ldots, \mathit{not}\,(\mathcal{C}_m : p_m).$$

where each $\mathcal{C}_i$ can be either a plain constant (as before) or an expression of the form $m_i(k_i)$ that we call *context designator*, which is a term where m_i can be seen as a(n arbitrary) meta-function indicating the required instantiation, and k_i is a constant that can be seen as analogous to a Skolem constant. Such term indicates the kind of context to which it must be substituted before bridge-rule execution, so it might be, for instance, $\mathit{university}(u)$, $\mathit{student\_data}(sd)$, $\mathit{treatment\_database}(d)$, $\mathit{diagnostic\_expert\_system}(de)$. There is no fixed format, rather it is intended as a designation of the required-for knowledge source, that can be either a knowledge repository or a reasoning module.

A bridge rule including context designators will be indicated as a *bridge rule pattern*, as it stands for its versions obtained by substituting the designators with actual contexts' names. Bridge-rule instantiation may be performed by an agent also by means of bridge-update rules, that are in charge of replacing designators with actual suitable knowledge sources. We assume that bridge-update rules' conclusions $\beta(\hat{x})$ are, in general, conjunctions, possibly including actions of the following distinguished forms:

(i) $\mathit{record}(\mathit{Item})$, which simply adds Item to A's knowledge base; Item can be either the "plain" bridge-rule result, or it can be obtained by processing such result via the evaluation of other atoms in $\beta(\hat{x})$;

(ii) $\mathit{incorporate}(\mathit{Item})$, which performs some more involved elaboration for incorporating Item into A's knowledge base. Notice that $\mathit{incorporate}$ is meant as a distinguished predicate, to be defined according to the specific application domain; in particular, it is intended to implement some proper form of belief revision.

(iii) $\mathit{instantiate}(S, m_i(k_i), L)$ which, for every bridge rule ρ with head matching with S, considers the context designator $m_i(k_i)$ and a list L of constants, and generates as many instances of ρ as obtained by substituting $m_i(k_i)$ (wherever it occurs) by elements of L. A bridge rules will be potentially applicable whenever all contexts in its body are constants, i.e., whenever all context designators, if present, have been replaced by actual contexts' names.

(iv) $\mathit{enable}(S, Q)$, which enables the application of a potentially applicable bridge rule ρ whose head matches with S and with associated trigger rule of the form Q **enables** S. It does so by generating its trigger, i.e., by adding Q as a new fact.

The combination of the introduction of both context designators and the $\mathit{instantiate}$ actions extends the expressiveness of the bridge-rule approach: even allowing variables in place of contexts' names would not allow for the specific customization performed here. The purpose of defining context designators as terms is that of avoiding the requirement of the involved domains to be finite. In fact, context designators can denote values in an infinite domain, where, however, a finite number of $\mathit{instantiate}$ actions generates a finite number of customized bridge rules. Notice that the computational complexity of the overall framework depends upon the computational complexity of the involved modules. In [8, 9] significant sample cases are reported.

5 Case Study: Bridge Rules Customization and Application

In order to explain the features that we have introduced so far we apply them to the case study. The agent acting on behalf of a prospective college student would for instance include the following trigger rule:

$$\begin{array}{l}\mathit{wish\_to\_enroll}(\mathit{Universities}, \mathit{Budget})\ \textbf{enables}\\ \qquad \mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL})\end{array}$$

The meaning is that the agent is supposed to be able to conclude at some stage of its operation $\mathit{wish\_to\_enroll}(\mathit{Universities}, \mathit{Budget})$, where $\mathit{Universities}$ is the list of universities which are of interest for the student, and Budget is the budget which is available for completing the application procedure. Whenever this conclusion is reached, the trigger rule is proactively activated, thus enabling a suitable bridge rule. This bridge rule exploits a quantitative reasoning module and might correspond to this simple bridge rule pattern, where however there is the relative context designator $\mathit{qr\_mod}(\mathit{mymod})$ to be instantiated.

$$\begin{array}{l}\mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL}) \leftarrow\\ \quad \mathit{qr\_mod}(\mathit{mymod}) : \mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL})\end{array}$$

Let us assume that the agent somehow (dynamically) instantiates this designator, e.g., to the name of a RASP module $\mathit{rasp\_mod}$, thus obtaining:

$$\begin{array}{l}\mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL}) \leftarrow\\ \quad \mathit{rasp\_mod} : \mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL})\end{array}$$

The RASP module, invoked via a suitable plugin, will return its results in $\mathit{Selected\_UniversitiesL}$, that will be a list representing the potential options for sending applications while staying within the given budget. A relevant role is performed by the corresponding bridge-update rule, which may have the form:

$$\begin{array}{l}\textbf{upon}\ \ \mathit{chooseU}(\mathit{Universities}, \mathit{Budget}, \mathit{Selected\_UniversitiesL})\ \textbf{then}\\ \quad \mathit{preferred\_subject}(\mathit{Subject}),\\ \quad \mathit{instantiate}(\mathit{apply}(\mathit{Univ}, \mathit{ResponseUniv}), \mathit{myuniv}(u), \mathit{Selected\_UniversitiesL}),\\ \quad \mathit{nearest\_sat\_center}(\mathit{Sc}),\ \ \mathit{nearest\_toefl\_center}(\mathit{Tc}),\\ \quad \mathit{instantiate}(\mathit{general\_tests}(\mathit{Subject}, \mathit{R1}, \mathit{R2}, \mathit{R3}), \mathit{sat\_center}(\mathit{sc}), [\mathit{Sc}]),\\ \quad \mathit{instantiate}(\mathit{general\_tests}(\mathit{Subject}, \mathit{R1}, \mathit{R2}, \mathit{R3}), \mathit{language\_center}(\mathit{lc}), [\mathit{Tc}]),\\ \quad \mathit{enable}(\mathit{general\_tests}(\mathit{Subject}, \mathit{R1}, \mathit{R2}, \mathit{R3}), \mathit{enabledgentest})\end{array}$$

By evaluating the sub-queries from left to right, as it is usual in Prolog, this rule will determine the preferred subject $\mathit{Subject}$, and via an $\mathit{instantiate}$ action it will create several copies of a bridge rule which finalizes the application (see below), namely one copy for each university included in $Selected\_UniversitiesL$. Notice that such bridge rules are not enabled yet. Then, the bridge-update rule finds the contexts' names Sc and Lc of nearest SAT and language-test centers respectively, where the student may perform the tests. The subsequent two *instantiate* actions, together with the *enable* action, will instantiate and trigger a suitable bridge rule pattern (shown below). The trigger part is, in particular:

$$\begin{array}{l} \mathit{enabledgentest}. \\ \mathit{enabledgentest}\ \textbf{enables}\ \ \mathit{general\_tests}(\mathit{Subject}, R1, R2, R3) \end{array}$$

which, as said, enables a bridge rule obtained by the following bridge rule pattern via its specialization to contexts' names Sc and Lc. This bridge rule will take care of performing the general tests (among which the language certification) and filling the general part of the application:

$$\begin{array}{rl} \mathit{general\_tests}(\mathit{Subject}, R1, R2, R3) \leftarrow & \mathit{sat\_center}(sc) : \mathit{general\_SAT\_test}(R1), \\ & \mathit{sat\_center}(sc) : \mathit{specific\_SAT\_test}(R2), \\ & \mathit{language\_center}(lc) : \mathit{language\_certification}(R3), \\ & \mathit{collegeorg} : \mathit{fill\_application} \end{array}$$

Each test will return its results, which are then dynamically recorded, whenever available, by the bridge-update rule:

$$\textbf{upon}\ \mathit{general\_tests}(\mathit{Subject}, R1, R2, R3)\ \textbf{then}\ \ \mathit{record}(\mathit{test\_res}(R1, R2, R3))$$

The recording of test results enables, via the following trigger rule, the application of the bridge rules, one for every selected university $Univ$, each of which will: send test the test results to that university; finalize the university-specific part of the application; wait for the response, returned in $ResponseUniv$.

$$\textbf{upon}\ \ \mathit{test\_res}(R1, R2, R3)\ \textbf{then}\ \ \mathit{apply}(\mathit{Univ}, \mathit{ResponseUniv})$$

The bridge rule pattern from which such bridge rules are obtained is:

$$\begin{array}{rl} \mathit{apply}(\mathit{Univ}, \mathit{ResponseUniv}) \leftarrow & \mathit{test\_res}(R1, R2, R3), \\ & \mathit{myuniv}(u) : \mathit{send\_test\_results}(R1, R2, R3), \\ & \mathit{myuniv}(u) : \mathit{complete\_application}(\mathit{ResponseUniv}) \end{array}$$

The corresponding bridge-update rules, of the form

$$\textbf{upon}\ \mathit{apply}(\mathit{Univ}, \mathit{ResponseUniv})\ \textbf{then}\ \ \mathit{record}(\mathit{response}(\mathit{Univ}, \mathit{Response}))$$

will record the responses, to allow a choice to be made among the universities that have returned a positive answer. Finally, enrollment must be finalized (code not shown here). Notice that, in the above bridge rules, some elements in the body implicitly involve the execution of specific actions (such as the payment of fees) that may take time to be executed, and may also involve user intervention (e.g., the student must personally and practically go to perform the SAT and TOEFL tests). Such actions have to be specified in the internal definition of the involved module(s), while user interventions emerge from the interaction between the agent and the user. For lack of space we do not discuss plan revision strategies (that might be needed in case of failure of some of the above steps), to be implemented via agent's reactive and proactive features.

6 Semantics

In order to account for heterogeneity of composing modules, in MCSs and then in DACMACSs and in ACEs each module is supposed to be based upon a specific logic. Reporting from [8], a logic L is a triple $(KB_L; Cn_L; ACC_L)$, where KB_L is the set of admissible knowledge bases of L. A knowledge base is a set of KB-elements, or "formulas". Cn_L is the set of acceptable sets of consequences, whose elements are data items or "facts". Such sets can be called "belief sets" or simply "data sets". $ACC_L : KB_L \rightarrow 2^{Cn_L}$ is a function which defines the semantics of L by assigning to each knowledge-base a set of acceptable sets of consequences.

For any of the aforementioned frameworks, consider an instance $\mathcal{A} = \langle A_1, \ldots, A_h \rangle$ composed of h distinct modules, each of which can be either the basic agents, or an event-action module, or a reasoning module, or an external context. Each module is seen as $A_i = (L_i; kb_i; br_i)$ where L_i is a logic, $kb_i \in KB_{L_i}$ is the module's knowledge base and br_i is a set of bridge rules. A data state of $\mathcal{A}$ is a tuple $S = (S_1, \ldots, S_h)$ such that each of the S_is is an element of Cn_i, i.e. a set of consequences derived from A_i's knowledge base according to the logic in which module A_i is defined.

When modules are not considered separately, but rather they are connected via bridge rules, desirable data states, called *equilibria*, are those where bridge-rule application is considered. In MCSs, equilibria are those data states S where each S_i is acceptable according to function ACC_i associated to L_i, taking however bridge rules application into account. Technically, a data state S is an equilibrium iff, for $1 \leq i \leq n$, it holds that $S_i \in ACC_i(mng_i(app(S), kb_i))$. This means that if one takes the knowledge base kb_i associated to module A_i, considers all bridge rules which are applicable in data state S (i.e., S entails their body), applies the rules, applies the management function, it obtains exactly S_i (or at least S_i is one of the possible sets of consequences). Namely, an equilibrium is a data state that encompasses the application of bridge rules. In dynamic environments however, this does not in general imply that a bridge rule is applied only once, and that an equilibrium, once reached, lasts forever (conditions for reachability of equilibria are discussed in literature, see [23] and the references therein). In fact, contexts are in general able to incorporate new data items, e.g, as discussed in [10], the input provided by sensors. Therefore, a bridge rule is in principle re-evaluated whenever a new result can be obtained, thus leading to evolving equilibria.

As DACMACS and ACEs are frameworks for defining agents and multi-agent systems, the interaction with the external environment and with other agents goes beyond simple sensor input and must be explicitly considered. This is done by assuming, similarly to what is done in Linear Temporal Logic, a discrete, linear model of time where each state/time instant can be represented by an integer number. States $t_0, t_1, \ldots$ can be seen as time instants in abstract terms, though in practice we have $t_{i+1} - t_i = \delta$, where δ is the actual interval of time after which we assume a given system to have evolved.

Consider then a notion of *updates*: for $i > 0$, let $\Pi_i = \langle \Pi_{iA_1}, \ldots, \Pi_{iA_h} \rangle$ be a tuple composed of finite updates performed to each module and let $\Pi = \Pi_1, \Pi_2, \ldots$ be a sequence of such updates performed at time instants $t_1, t_2, \ldots$. Let $\mathcal{U}_E$, for $E \in \{A_1, \ldots, A_h\}$, be the *update operator* that each module employs for incorporating the new information, and let $\mathcal{U}$ be the tuple composed of all these operators. Notice that each $\mathcal{U}_E$, i.e., each module-specific operator, encompasses the treatment of both self-generated updated and updated coming from interaction with an external environment.

In this more general setting data states evolve in time, where a *timed* data state at time T is a tuple $S^T = (S_1^T, \ldots, S_h^T)$ such that each S_i^T is an element of Cn_i at time T. The timed data state S^0 is an equilibrium according the MCSs definition. Later on however, transition from a timed data state to the next one, and consequently the definition of an equilibrium, is determined both by the update operators and by the application of bridge rules. A bridge rule ρ occurring in each composing module is now *potentially applicable* in S^T iff S^T entails its body. However, in the basic agent a potentially applicable bridge rule is applied only when it has been triggered by a trigger rule of the form seen above, i.e., if for some $T' \leq T$ we have that $S^{T'} \models Q$. In any event-action module M instead, a potentially applicable bridge rule is applied only if the module is *active*, i.e., if $S^{T'} \models tr_M$, where tr_M is an *event expression* which triggers the module evaluation (cf. [13]). Therefore, a timed data state of M at time $T+1$ is an equilibrium iff, for $1 \leq i \leq n$, it holds that $S_i^{T+1} \in ACC_i(mng_i(App(S^T), kb_i^{T+1}))$, where $kb_i^{T+1} = \mathcal{U}_i(kb_i^T, \Pi_T^i)$ and App is the extended bridge-rule applicability evaluation function. The meaning is that an equilibrium is now a data state which encompasses bridge rules applicability (with the new criteria) on the updated knowledge base. So, contexts now evolve in time, where we may say that $A_i^0 = (L_i; kb_i; br_i)$ as before, while $A_i^T = (L_i; kb_i^T; br_i)$. As discussed in [14], if both the update operators and the management functions preserve consistency of modules, then conditions for existence of an equilibrium (at some time T) are unchanged w.r.t. MCSs and DACMACS.

Notice that, for each bridge rule which is triggered (and so is applicable) at time T' the state when it is actually applied is not necessarily T', nor $T'+1$. In fact, a bridge rule becomes potentially applicable whenever a data state entail its body. So, the actual procedural sequence is the following:

- $S^{T'} \models Q$ for some trigger rule concerning bridge rule with conclusion $A(\hat{x})$, and then such a rule is executed at some time $T'' \geq T'$.
- At time $T \geq T''$ the results will be returned by the modules which are queried in the rule body; the case where $T' = T$, i.e., the bridge-rule body succeeds instantaneously, is an ideal extreme which is hardly the case in practice. In fact, internal and external modules may take some (a priori unpredictable) amount of time for returning their results.
- At time T, bridge-rule results will be elaborated by the management function, in our case implemented by the bridge-update rule.

The important aspect that allows us to smoothly incorporate enhanced ACE features in this semantics is that knowledge base updates in an agent are not necessarily determined from the outside. Rather, (part of) an update can also be the result of proactive self-modification. So, the generality and flexibility of ACE's semantics allows us to introduce advanced features without needing substantial modifications.

In particular, we consider bridge rule patterns as elements of agent's knowledge base. A bridge rule pattern will produce new bridge rules only when its context designators will be instantiated. Such instantiation can be seen as a part of a self-modification, i.e, it can be seen as an update. Therefore, for the main agent we now have $A_i^0 = (L_i; kb_i; br_i)$ and $A_i^T = (L_i; kb_i^T; br_i^T)$, where at each subsequent time the set of bridge rule associated to the module can be augmented by newly generated instances. The other definitions remain unchanged. This limited though effective semantic modifications constitute, in our opinion, a successful result of the research work that we present here. In fact, we obtain more general and flexible systems without significantly departing from the original MCSs' semantics, and this grants our approach a fairly general applicability.

7 Case Study: RASP Implementation

Below we discuss how to represent in RASP the case study discussed in Section 2. We do not report the full code, that the reader can find on the web site `http://www.dmi.unipg.it/formis/raspberry/` (section "Enrollment") where the solver Raspberry can also be obtained.[3] Our aim is to have a glance at how RASP works, and to demonstrate that the proposed approach is not only a more general architecture than basic ACE, but it has indeed a practical counterpart.

RASP code clearly must include a list of facts defining the universities to which the students is potentially interested, the SAT subjects (in general), and the SAT subjects corresponding to Courses (or Schools) available at each university.

```
% Universities
university(theBigUni).    university(theSmallUni).
university(thePinkUni).   university(theBlueUni).
university(theGreenUni).
% SAT subjects
sat_subject(literature).  sat_subject(mathematics).
sat_subject(chemistry).
% SAT subjects in each University
availableSubject(theBigUni, S) :- sat_subject(S).
availableSubject(theGreenUni, S) :- sat_subject(S).
availableSubject(theSmallUni, mathematics).
availableSubject(thePinkUni,  mathematics).
availableSubject(thePinkUni,  literature).
availableSubject(theBlueUni,  mathematics).
availableSubject(theBlueUni,  chemistry).
```

Below we then list: the tuition fees and the maximum fee allowed; the university rankings and the minimum required; the basketball team ranking, as it constitutes an additional evaluation factor.

```
% Tuition fees
tuitionFee(theBigUni,   21000).  tuitionFee(theSmallUni, 16000).
tuitionFee(thePinkUni,  15000).  tuitionFee(theBlueUni,  25000).
tuitionFee(theGreenUni, 15000).
% Constraint C1:  Tuition fee cannot exceed this threshold
maxTuition(22000).
```

[3] Raspberry, the grounder gringo (v.3.0.5), and the solver clasp (v.3.1.3) are used as follows:

```
raspberry_2.6.5 -pp -l3 -n 15000 -i enrollment_pref.rasp > enrollment_pref.asp
gringo-3.0.5 enrollment_pref.asp | clasp-3.1.3 0
```

```
% University reputation ranking R
reputation(theBigUni,   100).  reputation(theSmallUni,  90).
reputation(thePinkUni,   80).  reputation(theBlueUni,   75).
reputation(theGreenUni,  60).
% Constraint C2:  R must be higher than this threshold
reputationThrs(70).
% BasketballTeam Ranking
extraRank(theSmallUni, 10).  extraRank(theBigUni,   10).
extraRank(thePinkUni,   8).  extraRank(theBlueUni,   8).
extraRank(theGreenUni,  6).
```

The RASP fact below states that we have 1500 dollars, sum intended here as the budget available for completing applications. In general, symbol '#' indicates that an atom represents a resource. The constant before '#', here 'dollar', indicates the (arbitrary) name of the resource. The number after the '#' indicates an amount. In case of a fact, this amount is available initially, and can be then (in general) either consumed or vice versa incremented, as in RASP resource production can also be modeled.

```
% Budget for the application procedure
dollar#1500.
```

Now, the subject of interest and (if applicable) the status as foreign prospective students are indicated. Concerning the English language, nothing needs to be done if the student is not foreign, otherwise the TOEFL fee must be payed for performing the required test (we remind the reader that this RASP program evaluates the necessary expenses, so it is concerned with fees).

```
% My_subject
my_subject(mathematics).
% Omit the following fact if not foreign:
foreign.
% Language prerequisite
languageReqOK :- not foreign.
languageReqOK :- testTOEFLfee, foreign.
```

The universities where to potentially apply are derived according to the preferred subject, and the constraints concerning the university ranking and tuition fee. The student can apply if some university meeting the required requisites is actually found.

```
% Filtering of Universities
canApply(U,S) :- university(U), my_subject(S), reputation(U, R),
   availableSubject(U, S), reputationThrs(Th), R > Th,
   maxTuition(M), tuitionFee(U, Tu), Tu < M.
canApplyForSubject(Subj) :- canApply(Univ,Subj).
canApply :- canApply(Univ,Subject).
```

We now introduce proper RASP rules that perform quantitative reasoning, specifically by considering the fees for the different kinds of tests. The reader can ignore the prefix `[1-1]` which means that whenever the rule is applied, or "fired", this is done only once. This specification is not significant here, whereas it is useful in the description of more complex resource production/consumption processes.

```
% 1) General SAT test, fee1 fixed
  [1-1]: testSATfeeGen :- dollar#300, canApply.
% 2) Disciplinary SAT test, fee2 fixed
  [1-1]: testSATfeeSbj(mathematics) :-
           dollar#170, canApplyForSubject(mathematics).
  [1-1]: testSATfeeSbj(literature) :-
            dollar#180, canApplyForSubject(literature).
  [1-1]: testSATfeeSbj(chemistry) :-
           dollar#150, canApplyForSubject(chemistry).
  [1-1]: testSATfeeSbj(physics) :-
           dollar#160, canApplyForSubject(physics).
% 3) For foreign student, TOEFL fee3 fixed
  [1-1]: testTOEFLfee :- dollar#200, foreign, canApply.
% 4) Collegeorg application, fee4 fixed
  [1-1]: testCollegeOrg :- dollar#130, canApply.
```

A general rule with head `testGeneralDone` then establishes whether all general tests have been considered. If the available budget is too low and so no applications can issued, then no money is actually spent. Otherwise, the costs related to potential applications and the remaining amount (if any) are computed. Clearly, this code (omitted here) performs a quantitative evaluation and does not execute actual actions, which are left to the agent.

At this point, the Raspberry RASP solver can compute all solutions which maximize the number of applications. Solutions can be further customized with respect to the constraints. For instance, the standard `#maximize` ASP statements allow one to prefer universities with the best ranking and, in case of equivalent solutions, the ones with the best basketball team ranking (see the full code in the web site mentioned earlier, for the details on how to optimize the solution and enforce student preferences).

With the given facts, the best preferred solution provided by Raspberry involves applying to thePinkUni and theBigUni, with a total rating (sum of the two rankings) of 180 for the universities and 18 for the basketball teams.

If omitting maximization, there is a second solution which involves applying to thePinkUni and theSmallUni, with a total rating (sum of the two rankings) of 170 for the universities and 18 for the basketball teams.

The RASP module always returns the remaining (not spent) amount which is 90 dollars in the former case and 120 dollars in the latter one. Then, the agent might in general choose the best solution. However, it might instead choose another one based upon other criteria not expressed in the RASP program, i.e., geographic location or acceptance rates or maybe lesser expense, in case there would be relevant differences.

8 Concluding Remarks

The contribution of this paper is twofold. First, we have demonstrated, also by means of a practical example, how quantitative reasoning can be performed in agent-based frameworks. Second, we have enhanced modular approaches inspired to MCSs with partially specified bridge rules, that can be dynamically customized and activated according to agent's reasoning results. The approach of this paper is fairly general, and can be thus adapted to several application domains and to different agent architectures. Since no significant related work exists, our approach to coping with the cost of knowledge and the cost of action is relevant in a variety of domains, from logistics to configuration to planning, which are particularly well-suited for agents and MAS. An important application that we envisage is planning in robotic environments, where agents are embodied in robots that have limited resources available (first of all energy) and must complete their tasks within those limits, while possibly giving priority to the most important/urgent objectives.

References

[1] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In P. Cabalar and T. C. Son, editors, *Proc. of LPNMR 2013*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.

[2] M. Bienvenu, J. Lang, and N. Wilson. From preference logics to preference languages, and back. In *Proc. of KR 2010*, pages 414–424, 2010.

[3] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[4] R. H. Bordini and J. F. Hübner. BDI agent programming in AgentSpeak using *Jason*. In F. Toni and P. Torroni, editors, *CLIMA VI, selected papers*, volume 3900 of *LNCS*, pages 143–164. Springer, 2005.

[5] A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The KGP model of agency: Computational model and prototype implementation. In *Global Computing*, LNAI 3267, pages 340–367. Springer, 2005.

[6] G. Brewka, J. P. Delgrande, J. Romero, and T. Schaub. asprin: Customizing answer set preferences without a headache. In B. Bonet and S. Koenig, editors, *Proc. of AAAI-15*, pages 1467–1474. AAAI Press, 2015.

[7] G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proc. of AAAI-07*, pages 385–390. AAAI Press, 2007.

[8] G. Brewka, T. Eiter, and M. Fink. Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In M. Balduccini and T. C. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *LNCS*, pages 233–258. Springer, 2011.

[9] G. Brewka, T. Eiter, M. Fink, and A. Weinzierl. Managed multi-context systems. In T. Walsh, editor, *Proc. of IJCAI 2011*, pages 786–791. IJCAI/AAAI, 2011.

[10] G. Brewka, S. Ellmauthaler, and J. Pührer. Multi-context systems for reactive reasoning in dynamic environments. In T. Schaub, editor, *Proc. of ECAI-14*. IJCAI/AAAI, 2014.

[11] G. Brewka, I. Niemelä, and M. Truszczyński. Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4), 2008.

[12] A. Brik. *Extensions of Answer Set Programming*. PhD thesis, University of California, San Diego, 2012.

[13] S. Costantini. ACE: a flexible environment for complex event processing in logical agents. In M. Baldoni, L. Baresi, and M. Dastani, editors, *EMAS-15, Revised Selected Papers*, volume 9318 of *LNCS*. Springer, 2015.

[14] S. Costantini. Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In M. Truszczyński, G. Ianni, and F. Calimeri, editors, *Proc. of LPNMR-13*, volume 9345 of *LNCS*. Springer, 2015.

[15] S. Costantini and G. De Gasperis. Exchanging data and ontological definitions in multi-agent-contexts systems. In A. Paschke, P. Fodor, A. Giurca, and T. Kliegr, editors, *Proc. of RuleML 2015 Challenge*, CEUR Workshop Proceedings. CEUR-WS.org, 2015.

[16] S. Costantini and A. Formisano. Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of of Algorithms in Cognition, Informatics and Logic*, 64(1), 2009.

[17] S. Costantini and A. Formisano. Answer set programming with resources. *Journal of Logic and Computation*, 20(2):533–571, 2010.

[18] S. Costantini and A. Formisano. RASP and ASP as a fragment of linear logic. *Journal of Applied Non-Classical Logics*, 23(1-2):49–74, 2013.

[19] S. Costantini, A. Formisano, and D. Petturiti. Extending and implementing RASP. *Fundam. Inform.*, 105(1-2):1–33, 2010.

[20] S. Costantini and A. Tocchio. A logic programming language for multi-agent systems. In *Proc. of JELIA-02*, volume 2424 of *LNAI*. Springer, 2002.

[21] S. Costantini and A. Tocchio. The DALI logic programming agent-oriented language. In *Proc. of JELIA-04*, volume 3229 of *LNAI*. Springer, 2004.

[22] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[23] M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Distributed evaluation of nonmonotonic multi-context systems. *JAIR*, 52:543–600, 2015.

[24] M. Dastani, M. B. van Riemsdijk, and J. C. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer, 2005.

[25] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(12):308–334, 2004.

[26] A. Dovier, A. Formisano, E. Pontelli, and F. Vella. A GPU implementation of the ASP computation. In *Proc. of PADL 2016*, volume 9585 of *LNCS*, pages 30–47. Springer, 2016.

[27] M. Fisher. MetateM: The story so far. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3862 of *LNCS*, pages 3–22. Springer, 2005.

[28] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub. Progress in clasp series 3. In M. Truszczyński, G. Ianni, and F. Calimeri, editors, *Proc. of LPNMR-15*, volume 9345 of *LNCS*, pages 368–383. Springer, 2015.

[29] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*. Elsevier, 2007.

[30] K. V. Hindriks, W. van der Hoek, and J. C. Meyer. GOAL agents instantiate intention logic. In A. Artikis, R. Craven, N. K. Cicekli, B. Sadighi, and K. Stathis, editors, *Logic Programs, Norms and Action*, volume 7360 of *LNCS*, pages 196–219. Springer, 2012.

[31] G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *Proc. of KR 2012*, 2012.

[32] M. Maratea, L. Pulina, and F. Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.

[33] P. Naumov and J. Tao. Budget-constrained knowledge in multiagent systems. In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, *Proc. of AAMAS 2015*, pages 219–226. ACM, 2015.

[34] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[35] Web-references. Some ASP solvers. Clasp: `potassco.sourceforge.net`; Cmodels: `www.cs.utexas.edu/users/tag/cmodels`; DLV: `www.dlvsystem.com`; Smodels: `www.tcs.hut.fi/Software/smodels`.