Budget-Constrained Reasoning in Agent Computational Environments

(Extended Abstract)

Stefania Costantini
DISIM, Università di L'Aquila
Via Vetoio, I-67100 L'Aquila, Italy
stefania.costantini@univaq.it

Andrea Formisano
DMI, Università di Perugia — GNCS-INdAM
via Vanvitelli 1, I-06123 Perugia, Italy
formis@dmi.unipg.it

ABSTRACT

In this paper we consider the software-engineering problem of how to empower modular agent architectures with the capability to perform quantitative reasoning in a uniform and principled way.

Keywords

Multi-Context Systems; Quantitative Reasoning; Agent-Oriented Software Engineering

1. INTRODUCTION

There are many examples in agent-based applications where data might be potentially accessible or activities might be performed on the web, but agents are constrained by the available budget since access to knowledge bases is subject to costs. The work [8] identifies the problem that an agent faces when it has limited budget and costly queries to perform, and proposes a special resource-aware modal logic for deriving possible courses of actions, which however for complexity reasons can hardly be used in practice.

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. We consider the adoption of a flexible and reasonably efficient reasoning device that enables an agent which has costly objectives: (i) to establish which are the alternative possibilities within the available budget; (ii) to select, based upon its preferences, the goals to achieve and the resources to spend; (iii) to implement its choice. We illustrate the proposed approach via a realistic case study.

2. RESOURCE-BASED REASONING

In this paper, for quantitative reasoning we adopt RASP (Resource-based Answer Set Programming) [4, 5], which is an extension of ASP (Answer Set Programming), a well-established logic programming paradigm (cf., among many, [7]). RASP has been proven in [6] to be equivalent to an interesting fragment of Linear Logic, specifically an Horn fragment empowered however via the ASP default negation that Linear Logic does not provide. RASP explicitly introduces in ASP the notion of *resource*, and supports both formalization and quantitative reasoning on consumption and production of amounts of resources.

Appears in: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016), J. Thangarajah, K. Tuyls, C. Jonker, S. Marsella (eds.), May 9–13, 2016, Singapore.*

3. ENHANCING THE ACE FRAMEWORK

An enhanced Agent Computational Environment (ACE) [2] is defined as a tuple $\langle A, M_1, \ldots, M_r, C_1, \ldots, C_s, R_1, \ldots, R_q \rangle$ 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. The R_js (new addition w.r.t. [2]) are "Reasoning modules", which are specialized in specific reasoning tasks. The C_ks are contexts in the sense of MCSs (Multi-Context Systems, cf. [1]), i.e., external data/knowledge sources that the agent is able to locate and to query about some subject, but upon which it has no further knowledge and no control.

Interaction among ACE's components occurs via *bridge rules*, inspired by those of MCSs. However, here bridge rules become (via a smooth formal integration into ACE's semantics) *bridge rule patterns* of the following form:
$s \leftarrow (\mathcal{C}_1 : p_1), \ldots, (\mathcal{C}_j : p_j), not\,(\mathcal{C}_{j+1} : p_{j+1}), \ldots, not\,(\mathcal{C}_m : p_m).$
where each $\mathcal{C}_i$ can be either a constant indicating a context name, or a term of the form $m_i(k_i)$ that we call *context designator*, which indicates the *kind* of context (rather than the specific one) to be queried, to be specified before bridge-rule execution. Such a rule, once context designators have been instantiated to actual context names, is *applicable* (and s can thus be added to the consequences of a module's knowledge base) whenever each p_r, $r \leq j$, belongs to the consequences of module $\mathcal{C}_r$ while instead each p_w, $j < w \leq m$, does not belong to the consequences of $\mathcal{C}_w$.

In an ACE basic agent we adopt for bridge rules the agent-oriented modalities of DACMACSs [3]. First, bridge rules are applied proactively upon specific conditions, via *trigger rules* of the form Q **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, which is "fired" whenever Q evaluates to *true*. Second, the result returned by a bridge rule with head $A(\hat{x})$ will be exploited via a *bridge-update rule* of the form **upon** $A(\hat{x})$ **then** $\beta(\hat{x})$ where $\beta(\hat{x})$ specifies the elaboration to be applied for $\hat{x}$ to be incorporated into the agent's knowledge base.

4. CASE STUDY

As a case study we consider a student, 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. The example deals with two aspect of quantitative reasoning: (i) the cost of knowledge, as in practical terms a student applies in order *to know* whether she is admitted; (ii) reasoning under budget limits, as a student may send an application only if: she can afford the fees related to the application; in case of admission, she can then afford the tuition fees. The student 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.

Without any pretension to precision, we consider the following steps a student has to undergo: pass the general SAT test; pass the specific SAT test for the subject of interest; in case of foreign students, pass the TOEFL test; fill the general application on the application website (that we call collegeorg); send the SAT results to the universities of interest; complete the application for such universities. All steps are subject to the payment of fees, which are fixed (independent of the university where one applies) for the first four steps and depend upon each university for the last two steps.

In the case study, the agent will activate the following bridge rule, where *rasp_mod* is a RASP quantitative reasoning module which computes the list Sel_UnivL of the universities where it is possible to apply given the available budget:

$$\begin{aligned} &chooseU(Universities, Budget, Sel\_UnivL) \leftarrow \\ &\quad chooseU(Universities, Budget, Sel\_UnivL) : rasp\_mod \end{aligned}$$

The corresponding bridge-update rule will, given the preferred subject, instantiate and trigger a bridge rule pattern to perform the general tests, among which, if needed, the TOEFL test, and fill the general part of the application. It will then, given Sel_UnivL, generate for each element an instance of the following bridge rule pattern, to send test results and complete the application:

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

Finally, the student will choose where to enroll among the universities that return a positive answer (code not shown here).

5. RASP IMPLEMENTATION

Below we shortly illustrate the main features of the RASP module *rasp_mod*. (The full code and the RASP solver for its execution can be found at `www.dmi.unipg.it/formis/raspberry/`) As facts, we have the universities to which the students is interested, the SAT subjects (in general), and the SAT subjects available at each university.

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

Then we have: the tuition fees and the maximum fee allowed; the university rankings and the minimum required; the basketball team ranking, and the budget available for the applications (where notation "#" indicates a resource quantity, here money).

```
% Tuition fees
tuitionFee(theBigUni, 21000).    ...
% Tuition fee cannot exceed this threshold
maxTuition(22000).
% University reputation ranking R
reputation(theBigUni, 100).    ...
% Reputation must be higher than this threshold
reputationThrs(70).
% BasketballTeam Ranking
extraRank(theSmallUni, 10).    ...
% Budget for the application procedure
dollar#1500.
```

Then, the subject(s) of interest and (if applicable) the status as foreign student are indicated. The universities where to potentially apply (`canApply(Univ,Subject)`) are derived according to the preferred subject (`canApplyForSubject(Subject)`) and to the constraints concerning the university ranking and tuition fee. If any is found, then `canApply` becomes true. The RASP rules below perform quantitative reasoning, by considering the fees for the different kinds of tests.

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

If the available budget is too low, no applications can be issued. Otherwise, the costs related to potential applications and the remaining amount (if any) are computed. Clearly, this code performs a quantitative evaluation and does not execute actual actions, which are left to the agent (and to the user). The RASP solver can compute all solutions which maximize the number of applications given the constraints. Here, the best-preferred solution involves applying to thePinkUni and theBigUni, with a second-best solution which involves applying to thePinkUni and theSmallUni.

6. CONCLUDING REMARKS

In this paper we have demonstrated by means of a practical example how quantitative reasoning can be performed in modular agent-based frameworks. The approach of this paper is fairly general and novel, as no significant related work can be found so far.

REFERENCES

[1] G. Brewka, T. Eiter, and M. Fink. Nonmonotonic multi-context systems. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, Springer LNCS 6565, 2011.

[2] S. Costantini. ACE: a flexible environment for complex event processing in logical agents. In *Post-Proceedings of EMAS 2015 (Revised Selected Papers)*, Springer LNCS 9318, 2015.

[3] S. Costantini. Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In *Proc. of LPNMR 2015*, Springer LNCS 9345, 2015.

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

[5] S. Costantini and A. Formisano. Answer set programming with resources. *J. of Logic and Computation*, 20(2), 2010.

[6] S. Costantini and A. Formisano. RASP and ASP as a fragment of linear logic. *J. of Applied Non-Class. Logics*, 23(1-2), 2013.

[7] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*. Chapter 7. Elsevier, 2008.

[8] P. Naumov and J. Tao. Budget-constrained knowledge in multiagent systems. In *Proc. of AAMAS 2015*. ACM, 2015.