CoQuiAAS: Application of Constraint Programming for Abstract Argumentation *

Jean-Marie Lagniez, Emmanuel Lonca, and Jean-Guy Mailly

CRIL – U. Artois, CNRS
Lens, France
{lagniez,lonca,mailly}@cril.fr

Abstract. This paper is a description of our proposal to use Constraint Programming techniques to develop a software library dedicated to argumentative reasoning. We present a library which offers the advantages to be generic and easily adaptable.

1 Introduction

An abstract argumentation framework (AF) [1] is a directed graph where the nodes represent abstract entities called *arguments* and the edges represent *attacks* between these arguments. The meaning of such a graph is determined by an *acceptability semantics*, which indicates which properties a set of arguments must satisfy to be considered as a "solution" of the problem; such a set of arguments is then called an *extension*.

To compute usual reasoning tasks (credulous or skeptical acceptance of an argument, computation of an extension, enumeration of all the extensions) for the classical semantics (grounded, stable, preferred, complete), we propose to use Constraint Programming techniques, since this domain already proposes some very efficient solutions to solve high complexity combinatorial problems. We are in particular interested in propositional logic and some formalisms derived from it. More precisely, we use some CNF formulae to solve problems from the first level of the polynomial hierarchy, and some encodings in the *Partial Max-SAT* formalism for higher complexity problems. We take advantage of these encodings to solve these reasoning tasks, using some state-of-the-art approaches and software, which have proven their practical efficiency.

We have encoded those approaches for argumentation-based reasoning in a software library called CoQuiAAS. The aim of CoQuiAAS is dual. First, we provide some efficient algorithms to tackle the main requests for the usual semantics. Then, our framework is designed to be upgradable: one may easily add some new parameters (request, semantics), or realize new algorithm for the tasks which are already implemented. A first version is available on-line: http://www.cril.univ-artois.fr/coquiaas.

2 Example of Logical Encoding

We take advantage of the encodings proposed by [2] to design an approach to compute the extensions of an argumentation framework, and also to determine if an argument is

* This work benefited from the support of the project AMANDE ANR-13-BS02-0004 of the French National Research Agency (ANR).

skeptically or credulously accepted by an AF F. In this section, we only examplify our approach on the stable semantics. Our encodings are based on propositional logic, defined with the usual connectives on the set of Boolean variables $V_A = \{x_{a_i} \mid a_i \in A\}$. The propositional variable x_{a_i} denotes the fact the argument a_i is accepted by F. For a matter a readability, we use in the following a_i rather than x_{a_i}. Φ_{st}^F is a propositional formula from this language such that its models match the stable extensions of F.

In addition to computing a single extension and enumerating every extension, this encoding also allows us to answer the other requests for the stable semantics:

- Computing one (resp. each) stable extension of F is equivalent to the computation of one (resp. each) model of Φ_{st}^F.
- a_i is credulously accepted by F w.r.t. the stable semantics iff $\Phi_{st}^F \wedge a_i \not\vdash \bot$.
- a_i is skeptically accepted by F w.r.t. stable semantics iff $\Phi_{st}^F \wedge \neg a_i \vdash \bot$.

A similar reasoning scheme from Besnard and Doutre encoding of complete semantics lead us to define a procedure for each reasoning task under the complete semantics. It is the case that unit propagation on the encoding of complete semantics allows to perform grounded reasoning, while preferred reasoning requires a transformation of the complete semantics encoding into a Partial Max-SAT instance such that each preferred extension correspond to one of its Maximal Satisfiable Set (MSS).

3 CoQuiAAS : Design of the Library

We have chosen the language C++ to implement CoQuiAAS to take advantage of the Object Oriented Programming (OOP) paradigm and its good computational efficiency. First, the use of OOP allows us to give CoQuiAAS an elegant conception, which is well suited to maintain and upgrade the software. Moreover, C++ ensures having high computing performances, which is not the case of some other OOP languages. At last, it makes easier the integration of coMSSExtractor, a C++ underlying tool we used to solve the problems under consideration.

coMSSExtractor [3] is a software dedicated to extract MSS/coMSS pairs from a Partial Max-SAT instance. As coMSSExtractor integrates the Minisat SAT solver [4] – which is used as a black box to compute MSSes – the API provided by coMSSExtractor allows us to use the API provided by Minisat to handle the requests that require a simple SAT solver. This way, CoQuiAAS does not need a second solver to compute the whole set of requests it is attended to deal with.

The core of our library is the interface *Solver*, which contains the high-level methods required to solve the problems. The method `initProblem` makes every required initialization given the input datas. In the case of our approaches, it initializes the SAT solver or the coMSS extractor with the logical encoding corresponding to the AF, the semantics and the reasoning task to perform. The initialization step depends on the concrete realization of the *Solver* interface returned by the **SolverFactory** class, given the command-line parameters of CoQuiAAS. The method `computeProblem` is used to compute the result of the problem, and `displaySolution` prints the result into the dedicated output stream using the format expected by the competition.

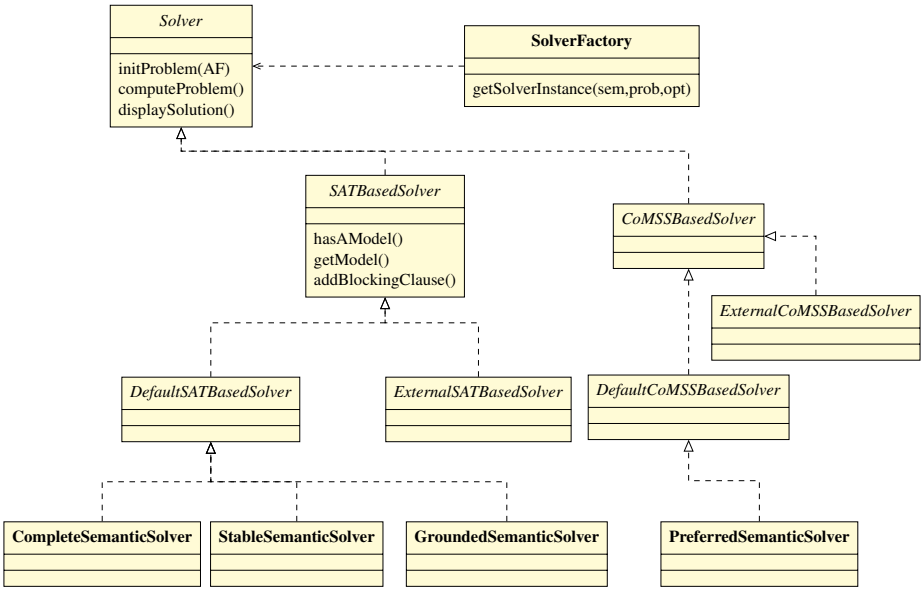

Fig. 1. Simplified class diagram of the solver part of CoQuiAAS

The abstract class *SATBasedSolver* (respectively *CoMSSBasedSolver*) gathers the features and initialization common to each solver based on a SAT solver (respectively a coMSS extractor), for instance the method `hasAModel` which returns a Boolean indicating if the SAT instance built from the argumentation problem is consistent or not. Among the subclasses of *SATBasedSolver*, we built *DefaultSATBasedSolver* and its subclasses, which are dedicated to use the API of coMSSExtractor to take advantage of its SAT solver features, inherited from Minisat, to solve the problems. If the user wants to call any other SAT solver rather than coMSSExtractor – as soon as the given semantics is compatible with SAT encodings – a command-line option leads the **SolverFactory** to generate an instance of the class *ExternalSATBasedSolver*, which also extends *SATBasedSolver*. This solver class is initialized with a command to execute so as to call any external software able to read a CNF formula written in the DIMACS format, and to print a solution using the format of SAT solvers competitions. This class allows to execute the command provided to CoQuiAAS to perform the computation related to the problem. This feature enables, for instance, the comparison between the relative efficiency of several SAT solvers on the argumentation instances. The same pattern is present in the coMSS-based part of the library, with the class *CoMSSBasedSolver*, which can be instantiated *via* the default solver *DefaultCoMSSBasedSolver*, which uses coMSSExtractor, or *via* the class *ExternalCoMSSBasedSolver* to use any external software whose input and output correspond to coMSSExtractor ones, for the pairs request/semantics corresponding to our coMSS-based approaches.

Our design is flexible enough to make CoQuiAAS evolutive. For instance, it is simple to create a solver based on the API of another SAT solver than coMSSExtractor:

creating a new class **MySolver** which extends *SATBasedSolver* (and also, the interface, *Solver* which is the root of each solver) and implementing the required abstract methods (`initProblem`, `hasAModel`, `getModel` and `addBlockingClause`) is the only work needed. It is also possible to extend directly the class *Solver* and to implement its methods `initProblem`, `computeProblem` and `displaySolution` to create any kind of new solver. For instance, if we want to develop a CSP-based approach for argumentation-based reasoning, using encodings such that those from [5], we just need to add a new class *CSPBasedSolver* which implements the interface *Solver*, and to reproduce the process which lead to the conception of the SAT-based solvers, but using this time the API of a CSP solver (or an external CSP solver).

Once the solver written, we just need to give an option which executes CoQuiAAS, and to update the method `getSolverInstance` in the **SolverFactory**, which knows the set of the command-line parameters (stored in the map `opt`). For instance, the parameter `-solver MySolver` can be linked to the use of the class *MySolver* dedicated to the new solver. The code given below is sufficient to do that.

```
if(opt["-solver"] == "MySolver") return new MySolver(...);
```

In the way we conceived the interface *Solver*, it is supposed that a solver is dedicated to a single problem and a single semantics. Thus, it is possible to implement a class which executes a unique algorithm, suited to a single pair (problem,semantics). For instance, [6] describes a procedure which determines if a given argument belongs to the grounded extension of an AF. We can consider the possibility to implement a class **GroundedDiscussion** which realizes the interface **Solver** to solve the skeptical decision problem under the grounded semantics using this dedicated algorithm.

This default behaviour of CoQuiAAS does not prevent the implementation of classes able to deal with several request for a given semantics, as soon as the **SolverFactory** returns an instance of the right solver for the considered semantics. Thus, thanks to the possibility to tackle each problem for a given semantics through a SAT instance (or a MSS problem), we have simplified the design of our solvers using a single class for each semantics, taking advantage of the *template* design pattern.

References

1. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and n-person games. Artificial Intelligence **77**(2) (1995) 321–357
2. Besnard, P., Doutre, S.: Checking the acceptability of a set of arguments. In: NMR. (2004) 59–64
3. Grégoire, É., Lagniez, J.M., Mazure, B.: An experimentally efficient method for (MSS, CoMSS) partitioning. In: AAAI. (2014) 2666–2673
4. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT. (2003) 502–518
5. Amgoud, L., Devred, C.: Argumentation frameworks as constraint satisfaction problems. Annals of Mathematics and Artificial Intelligence **69**(1) (2013) 131–148
6. Caminada, M., Podlaszewski, M.: Grounded semantics as persuasion dialogue. In: COMMA. (2012) 478–485