$\mathcal{A}$ -system: Declarative Programming with Abduction^{*}

Bert Van Nuffelen¹ and Antonis Kakas²

¹ Dept. Computer Science, K.U.Leuven
Celestijnenlaan 200A, Leuven, Belgium
bertv@cs.kuleuven.ac.be

² Dept. of Computer Science, University of Cyprus
75 Kallipoleos St. Nicosia, Cyprus
antonis@ucy.ac.cy

1 General Information

The $\mathcal{A}$ -system [4] is a new system for performing abductive reasoning within the framework of Abductive Logic Programming (ALP). The principles behind the system are founded by work on two earlier systems ACLP [2, 3] and SLDNFA(C) [1, 6]. The basic inference mechanism of the system combines abductive logic programming and constraint logic programming. In its computation it reduces the high level specification of the problem and goal at hand to a lower level constraint store. This constraint store is managed by an efficient constraint solver returning information to the abductive reduction process in order to help this in its search for a solution.

In the development of the $\mathcal{A}$ -system particular attention was put into general purpose control and search strategies in order to enhance the computational behaviour of the system. The main idea is to suspend each commitment to a choice up to the moment where no other information can be derived in a deterministic way. This improves it over its ancestors.

The $\mathcal{A}$ -system is implemented as a meta interpreter on top of Sicstus Prolog where at least version 3.8.5 is needed. The system is therefore available on each platform on which Sicstus is available.

2 Description of the System

The $\mathcal{A}$ -system is a declarative problem solving environment. It allows us to specify the problem domain in a well structured way in terms of an abductive logic program (ALP). A problem specification consists of two parts of information: definitional knowledge and assertional knowledge or integrity constraints. In the definitional part predicates are defined in a (constraint) logic program by specifying the rules embodying all the information known of these predicates.

In a planning domain example the predicate `holds_at(P, T)` is defined as:

^{*} <http://www.cs.kuleuven.ac.be/~dtai/kt>

```
holds_at(P,T):- initially(P), not(init_clipped(P,T)).
holds_at(P,T):- initiates(P,A), act(A,E), E<T, not(clipped(E,P,T)).
```

The predicates which are not defined like `act(A,E)` are called *open or abducible*.

Integrity constraints are universal quantified statements which express a property of the theory (application domain) that must remain true whenever this theory is extended with a (partial) definition of the open predicates. In practice, this means that the integrity constraints encode at a declarative level properties of the abductive solutions to the goals of our problem domain. In the $\mathcal{A}$ -system they have to be specified as denials with as head `ic`. For example the two constraints below ensure that the second argument of `act(A,T)` is within the valid time interval; and that the preconditions of each action is satisfied.

```
ic:- act(A,T),not(time(T)).
ic:- act(A,T),not(pre_conditions_hold(A,T)).
```

Given a query Q the $\mathcal{A}$ -system computes an extension of the open predicates such that this set entails Q and the integrity constraints. Semantically this solution is formalised as an abductive explanation of Q . Computationally this is found through a reduction process where the system goes through a cycle of selecting a choice point, making this choice and propagating deterministically this choice as much as possible. In general, the $\mathcal{A}$ -system postpones choices as long as possible and tries to evaluate the choice points in an informed way. This is mainly done by interacting with the constraint solver.

3 Applying the System

A simple preprocessor accepts the ALP problem representation and compiles it into a readable format for the system. The preprocessor will load automatically the compiled specification. The $\mathcal{A}$ -system can then be queried by the call `asystem_solve(query(Args))`. The system returns as answer a table of atoms of the open predicates whose addition to the program will entail the query.

3.1 Methodology

The $\mathcal{A}$ -system has been applied on a number of applications. Although most of them are not of "industrial scale" some general guidelines have emerged from these experiments, which can be followed to build an application.

The main advantage of the $\mathcal{A}$ -system is the modular development of the problem representation that it allows. This stems from the fact that this representation can be a direct mapping of the declarative high-level specification of the problem. The $\mathcal{A}$ -system thus allows us to build up the problem representation incrementally. Typically, this starts with a choice of the alphabet. The alphabet will determine the way the specification will look and will potentially influence the reasoning efficiency of the system. For example, in the block world planning, in the simplest case, it is sufficient to capture the movement of a block from one

place to another in one action `move(X,Y,T)`. This specification cannot represent a state in which the robot is holding a block. If that becomes important the move action can be split into `pick(X,T)` and `put(X,Y,T)`. This refinement is done locally affecting only part of the specification. In general, such refinements allow the user to solve first a more abstract problem whose solutions can then be further reduced to give concrete actual solution of the problem e.g. as in the process of hierarchical planning.

Another way of refining the declarative problem representation in ALP is to add extra integrity constraints. Due to this extra knowledge the $\mathcal{A}$ -system will (in most cases) be able to prune unwanted branches earlier or eliminate uninteresting solutions from the more general specification. The compactness of the ALP representations also means that the framework is well suited for problems in which the specification is subject to (regular) changes. The adaptation to changes can be done easily without disturbing the whole specification.

3.2 Users and Useability

A user familiar with logic programming or first order logic will be able to use the $\mathcal{A}$ -system with little extra help. A good knowledge about the problem domain will result in better and more informed specifications and thus in better performance. The $\mathcal{A}$ -system also allows the user to influence its behaviour using some parameters on how this would interact with the underlying constraint solver. For this reason it is useful to be familiar with some details of CLP.

The modeling language poses very few restrictions on the problem domains on which it can be applied. However, the current implemented prototype has some. At this moment only problems which can be modeled inside the finite domains framework can be solved efficiently. This is because only the finite domain CLP solver is integrated in the system. In principle there are no limitations to integrating CLP solvers over other domains. Furthermore, we are extending the system with aggregates (e.g. cardinality, sum, and average) so that a wider range of problem domains can be solved. This extension will allow the $\mathcal{A}$ -system to solve for example problems of planning with resources.

4 Evaluating the System

The $\mathcal{A}$ -system has been tested on constraint satisfaction problems (CSP) (e.g. N-Queens, Graph Coloring and Scheduling problems), examples of diagnosis and standard planning problems taken from the AI Planning Systems Competition 2000. Most extensively it has been tested on planning as in this domain the problems cannot be reduced in a deterministic way to a constraint store but the system must search for a solution. In simple CSP problems like the n-queens or graph coloring and scheduling problems the whole specification can be reduced to a CLP-constraint store. The real problem solving is then left to CLP solver. However in the planning domain, the search for a solution is an interleaved

process of making a choice by the high level procedure of the $\mathcal{A}$ -system and utilizing the information about the impact of this choice on the constraint store.

As mentioned above, currently only a finite domain solver is incorporated in the system, and thus the problem size is limited by the efficiency of the constraint solver as this size increases. As the problem size increases the system, due to non optimal data structures that it uses, may slow down on large problem instantiations. It has been successfully tested in the blocks world domain (using the move operator) up to 100 blocks. On other planning domains like logistics the scaling was not as good. In general, the problem size which can be handled depends on the generated constraint store. If this store becomes too complex for the constraint solver, the derivation process might get stuck in a local satisfiability check of this.

4.1 Benchmarks and Comparison

Currently very few benchmarks are available for testing the full capabilities of the $\mathcal{A}$ -system with the AIPS planning competition test set being the best example. The $\mathcal{A}$ -system has been evaluated in [4] with some of the problems from this set.

A comparison of the $\mathcal{A}$ -system with other systems can be separated in two classes: to other general purpose systems like Smodels and DLV or more specialized systems like AI planners. At this moment no extensive study has been made to compare the $\mathcal{A}$ -system with the second class of systems. In the first class, the $\mathcal{A}$ -system is capable to solve the same type of applications. However, Smodels and DLV which work on propositional theories are more robust on some problems making extensive use of heuristics in their bottom up computation. Currently, we are experimenting with different general heuristics that would allow the $\mathcal{A}$ -system to avoid infinitely growing search branches and thus make it more robust. A number of recent comparison experiments [5] indicate that the top down reducing process of the $\mathcal{A}$ -system performs, on some classes of problems, better than Smodels. A characteristic for such a class is that the high-level specification can be deterministically reduced into a constraint store.

References

1. Marc Denecker and Danny De Schreye. SLDNFA: an abductive procedure for normal abductive programs. *J. Logic Programming*, 34(2):111–167, 1998.
2. A. C. Kakas and C. Mourlas. ACLP: Flexible solutions to complex problems. In *Proceedings of LPNMR-97*, pages 387–398, 1997.
3. A.C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *Journal of Logic Programming*, 44(1-3):129–177, July, August 2000.
4. A.C Kakas, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. In *Proceedings of IJCAI'01*, 2001. to appear.
5. Nikolay Pelov, Emmanuel De Mot, and Maurice Bruynooghe. A comparison of logic programming approaches for representation and solving of constraint satisfaction problems. In *Proceedings of NMR'2000*, pages 1–15, 2000.
6. Bert Van Nuffelen and Marc Denecker. Problem solving in ID-logic with aggregates. In *Proceedings of NMR'2000, special track on Abductive Reasoning*, pages 1–9, 2000.