The $\mathcal{AC}(C)$ Language: Integrating Answer Set Programming and Constraint Logic Programming

Forrest Sheng Bao
Department of Computer Science
Texas Tech University
Lubbock, Texas 79409
forrest.bao @ gmail.com

Abstract
Combining Answer Set Programming (ASP) and Constraint Logic Programming (CLP) can create a more powerful language for knowledge representation and reasoning. The language $\mathcal{AC}(C)$ is designed to integrate ASP and CLP. Compared with existing integration of ASP and CSP, $\mathcal{AC}(C)$ allows representing user-defined constraints. Such integration provides great power for applications requiring logical reasoning involving constraints, e.g., temporal planning. In $\mathcal{AC}(C)$, user-defined and primitive constraints can be solved by a CLP inference engine while the logical reasoning over those constraints and regular logic literals is solved by an ASP inference engine (i.e., solver). My PhD work includes improving the language $\mathcal{AC}(C)$, implementing its faster inference engine and investigating how effective the new system can be used to solve a challenging application, temporal planning.

Goal of the Work
Answer Set Programming (ASP) and Constraint Logic Programming (CLP) are two paradigms in Logic Programming. ASP has a Prolog-like syntax and stable model semantics supporting nonmonotonicity (Gelfond 2008). It has been used to build reasoning systems in many fields. An ASP program consists of rules in the form:

$$ l_0 : - l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n $$

where $l_i$ for $i \in [0..n]$ is a regular logic literal and the not is called negation as failure. CLP extends logic programming to include the concept of constraint satisfaction where any literal is either a primitive constraint or a user-defined constraint over the domain (e.g., real numbers). For example, we can define constraints \textit{clear} and \textit{apart} as

$$ \text{clear}(T) : - 8.5 < T < 11.75, $$
$$ \text{clear}(T) : - 14 < T < 16, $$
$$ \text{apart}(T_1, T_2, G) : - T_1 - T_2 \geq G, $$

where $\text{clear}(T)$ can mean “the sky is clear at time $T$” and $\text{apart}(T_1, T_2, G)$ can mean “two times $T_1$ and $T_2$ are apart enough by a gap $G$.” The first two rules can be used to express the idea that “the sky is clear at any time $T$ in $8:30AM - 11:45AM$ or $2PM - 4PM.” The last rule can be used to express the idea that “$T_1$ and $T_2$ are apart enough, if they are at least $G$ hour(s) apart and the sky is clear at both $T_1$ and $T_2$.” Here user-defined constraint \textit{clear} is used to build another user-defined constraint \textit{apart}.

If we combine ASP and CLP together, we can build a more powerful language for applications involving constraints in logical reasoning, such as temporal planning. For example, we may easily express “two flights are both good to go, if their take-off times are at least 6 minutes apart, and neither of them is known canceled” as

$$ \text{good2go(FltA, FltB)} : - \text{apart} (\text{takeoff}(\text{FltA}), \text{takeoff}(\text{FltB}), 0.1) \text{not canceled} (\text{FltA}), \text{not canceled} (\text{FltB}), \text{FltA} = \text{FltB}. $$

The function $\text{takeoff}()$ is called a “bridge function” that maps a regular logic term (here, $\text{FltA}$ or $\text{FltB}$) into a value in constraint domain (here, a real number). The last 3 literals are regular ASP literals, where we also employ the nonmonotonicity of ASP.

My PhD work will be focusing on the language $\mathcal{AC}(C)$ that combines ASP and CLP, and its inference engine (i.e., solver). A previous work in my lab has introduced an early version of $\mathcal{AC}(C)$ (Mellarkod, Gelfond, and Zhang 2008). But there are a lot to improve. The existing $\mathcal{AC}(C)$ solver does not support all features of $\mathcal{AC}(C)$ and has several restrictions for programmers (Mellarkod 2007). The existing solver is built on an ASP solver and a CLP inference engine, where the query interface between ASP solver and CLP inference engine is not efficient.

Outcomes are expected on three aspects. On the language aspect, the syntax and semantics of $\mathcal{AC}(C)$ will be revised. Necessary syntactic constructs will be added to simplify the programming. On the solver aspect, the new solver will support more existing $\mathcal{AC}(C)$ features that the previous one is not able to. The new solver will also support new features of $\mathcal{AC}(C)$. The algorithm of the new solver will be made more efficient by using incremental querying between ASP and CLP inference engines. A solving framework may be built such that different ASP and CLP inference engines can be easily hooked up to work together. On the application
aspect, \(\mathcal{AC}(C)\) will be used to represent and solve temporal planning problems, which may be hard to solve in other tools. The methodology of using \(\mathcal{AC}(C)\) to model problems will also be examined in this process.

**Plan for Research**

The first step is to revise the \(\mathcal{AC}(C)\) language. Through a work converting PDDL2.1-described temporal planning problems into \(\mathcal{AC}(C)\)-representation (Bao et al. 2010), I have had a better understanding to the design of the language and the methodology of modeling problems in \(\mathcal{AC}(C)\). The understanding will be reflected in the revision to \(\mathcal{AC}(C)\). As mentioned earlier, syntactic constructs will be added to make programming easier. When revising the language, I want to hear the comments from declarative programming users. The grammar used to write \(\mathcal{AC}(C)\) programs for \(\mathcal{AC}(C)\) solver to solve, will be released as well. It will be designed as “user-friendly” as possible to lower the entry level for programmers without much logic programming background.

Understanding existing \(\mathcal{AC}(C)\) solver will be the next step. As a rule of thumb, knowing how previous systems are built can help build a better system. Besides, the current solver cannot support many features of \(\mathcal{AC}(C)\). For example, negations (classical or default) are not allowed in front of atoms that are not regular ASP ones. When translating PDDL2.1 into \(\mathcal{AC}(C)\), we have noticed many inconveniences due to the solver. Furthermore, I will identify bottlenecks in the existing solver.

Then, I will start building a new \(\mathcal{AC}(C)\) solver from “hacking” the existing solver. The new solver will support more \(\mathcal{AC}(C)\) features (including those supposed to be supported by the existing \(\mathcal{AC}(C)\) solver), and will be more efficient. Since we will have a revision of the language \(\mathcal{AC}(C)\), the solver will be modified accordingly. To improve solving efficiency, a new technique, incremental querying, will be implemented. The ASP solver cannot evaluate a constraint (primitive or user-defined) literal but can know whether it is required to be true or false by methods such as unit propagation. Once the ASP solver requires a constraint literal to be true, it queries the CLP inference engine to check the satisfiability of the literal. Due to backtracking, a constraint literal may be queried many times in solving one \(\mathcal{AC}(C)\) program. If we can implement incremental querying, e.g., only query new queries, the efficiency of the solver can be greatly improved.

After building a new working solver from existing solver, I may implement a general framework for building \(\mathcal{AC}(C)\) solver from various ASP and CLP inference engines. Current \(\mathcal{AC}(C)\) solver, is not built on state-of-the-art ASP and CLP inference engines. To take advantage of the development in ASP and CLP solving techniques, a general framework is need to allow ASP and CLP inference engines, that are under active development and maintenance, to easily hook up into an \(\mathcal{AC}(C)\) solver. For ASP part, I may choose Clasp which is the fastest ASP/SAT solver so far. For CLP part, I may choose one from swi-prolog and YAP.

Finally, I will use \(\mathcal{AC}(C)\) to represent and solve challenging problems that have great drawbacks to be solved in other tools. In this process, I can also test the modeling ability and expressiveness of the new \(\mathcal{AC}(C)\). A good test bed is temporal planning. The language PDDL is a standard in the planning community to define planning problems. Temporal planning problems have been supported since PDDL2.1. Existing planners (Huang, Chen, and Zhang 2010; Hu 2007; Coles et al. 2008) for temporal planning problems have several drawbacks, as analyzed in (Bao et al. 2010). Many planners translate a PDDL2.1 program into a programming problem, e.g., CSP (Hu 2007) and SAT (Huang, Chen, and Zhang 2010), and use the solver for the programming problem as the planner. Following this idea, I will translate PDDL2.1 into the language \(\mathcal{AC}(C)\) and then use \(\mathcal{AC}(C)\) solver to find the plan. The preliminary translation from PDDL2.1 to \(\mathcal{AC}(C)\) has been proposed and this approach is hopefully to solve the drawbacks of existing approaches (Bao et al. 2010). An automated PDDL2.1-to-\(\mathcal{AC}(C)\) translator and an \(\mathcal{AC}(C)\) solver can form an \(\mathcal{AC}(C)\)-based temporal planner.

**Progress so far**

We have shown in a published work a preliminary design of a translation from PDDL2.1 to \(\mathcal{AC}(C)\) (Bao et al. 2010). The syntax and semantics of the language \(\mathcal{AC}(C)\) itself have been reviewed in studying the translation. We have investigated new features that should be added into \(\mathcal{AC}(C)\) and features that should be supported by the new solver. We also have an initial design of the incremental algorithms.

**References**


