Clingo = ASP + Control: Preliminary Report

Martin Gebser\textsuperscript{1,2}, Roland Kaminski\textsuperscript{2}, Benjamin Kaufmann\textsuperscript{2}, and Torsten Schaub\textsuperscript{2}\textsuperscript{*}
\textsuperscript{1}Aalto University, Finland \quad \textsuperscript{2}University of Potsdam, Germany
submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

We present the new ASP system clingo 4. Unlike its predecessors, being mere monolithic combinations of the grounder gringo with the solver clasp, the new clingo 4 series offers high-level constructs for realizing complex reasoning processes. Among others, such processes feature advanced forms of search, as in optimization or theory solving, or even interact with an environment, as in robotics or query-answering. Common to them is that the problem specification evolves during the reasoning process, either because data or constraints are added, deleted, or replaced. In fact, clingo 4 carries out such complex reasoning within a single integrated ASP grounding and solving process. This avoids redundancies in relaunching grounder and solver programs and benefits from the solver’s learning capacities. clingo 4 accomplishes this by complementing ASP’s declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, clingo 4 offers a new directive that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages. By strictly separating logic and control programs, clingo 4 also abolishes the need for dedicated systems for incremental and reactive reasoning, like iclingo and oclingo, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

1 Introduction

Standard Answer Set Programming (ASP; (Baral 2003)) follows a one-shot process in computing stable models of logic programs. This view is best reflected by the input/output behavior of monolithic ASP systems like dlv (Leone et al. 2006) and clingo (Gebser et al. 2011b). Internally, however, both follow a fixed two-step process. First, a grounder generates a (finite) propositional representation of the input program. Then, a solver computes the stable models of the propositional program. This rigid process stays unchanged when grounding and solving with separate systems. In fact, up to now, clingo provided a mere combination of the grounder gringo and the solver clasp. Although more elaborate reasoning processes are performed by the extended systems iclingo (Gebser et al. 2008) and oclingo (Gebser et al. 2011a) for incremental and reactive reasoning, respectively, they also follow a pre-defined control loop evading any user control. Beyond this, however, there is substantial need for specifying flexible reasoning processes, for instance, when it comes to interactions with an environment, as in assisted living, robotics, or with users, advanced search, as in multi-objective optimization, planning, theory solving, or heuristic search, or recurrent query answering, as in hardware analysis and testing or stream processing. Common to all these advanced forms of reasoning is that the problem specification evolves during the reasoning processes, either because data or constraints are added, deleted, or replaced.

\textsuperscript{*} Affiliated with the Simon Fraser University, Burnaby, Canada, and Griffith University, Brisbane, Australia.
The new clingo 4 series offers novel high-level constructs for realizing such complex reasoning processes. This is achieved within a single integrated ASP grounding and solving process in order to avoid redundancies in relaunching grounder and solver programs and to benefit from the learning capacities of modern ASP solvers. To this end, clingo 4 complements ASP’s declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, clingo 4 offers a new directive #program that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages embedded via the #script directive. For exercising control, the latter benefit from a dedicated clingo library that does not only furnish grounding and solving instructions but moreover allows for continuously assembling the solver’s program in combination with the directive #external. Hence, by strictly separating logic and control programs, clingo 4 abolishes the need for special-purpose systems for incremental and reactive reasoning, like iclingo and oclingo, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

2 Controlling grounding and solving in clingo 4

A key feature, distinguishing clingo 4 from its predecessors, is the possibility to structure (non-ground) input rules into subprograms. To this end, the directive #program comes with a name and an optional list of parameters. Once given in the clingo 4 input, it gathers all rules up to the next such directive (or the end of file) within a subprogram identified by the supplied name and parameter list. As an example, two subprograms base and acid(k) can be specified as follows:

```plaintext
1 a(1).
2 #program acid(k).
3 b(k).
4 #program base.
5 a(2).
```

Note that base, with an empty parameter list, is a dedicated subprogram that, in addition to rules in the scope of a directive like the one in Line 4, gathers all rules not preceded by a #program directive. Hence, in the above example, the base subprogram includes the facts a(1) and a(2). Without further control instructions (see below), clingo 4 grounds and solves the base subprogram only, essentially yielding the standard behavior of ASP systems. The processing of other subprograms, such as acid(k) with the schematic fact b(k), is subject to scripting control.

For a customized control over grounding and solving, a main routine (taking a control object representing the state of clingo 4 as argument) can be specified in either of the embedded scripting languages Lua and Python. For illustration, let us consider two Python main routines:

```plaintext
6 #script (python)
7 def main(prg):
8     prg.ground("base",[])
9     prg.solve()
10 #end.
```

While the control program on the left matches the default behavior of clingo 4, the one on the right ignores all rules in the base program but rather, in Line 8, contains a ground instruction for acid(k), where the parameter k is instantiated with the term 42. Accordingly, the schematic fact b(k) is turned into b(42), and the solve command in Line 9 yields a stable model consisting of b(42) only. Note that ground instructions apply to the subprograms given as
arguments, while `solve` triggers reasoning w.r.t. all accumulated ground rules. In fact, a `solve` command makes `clingo 4` instantiate pending subprograms and then perform reasoning. That is, when Line 9 is replaced, e.g., by `print 'Hello!'`, `clingo 4` merely writes out `Hello!` but does neither ground any subprogram nor compute stable models.

In order to accomplish more elaborate reasoning processes, like those of `iclingo` and `oclingo` or customized ones, it is indispensable to activate or deactivate ground rules on demand. For instance, former initial or goal state conditions need to be relaxed or completely replaced when modifying a planning problem, e.g., by extending its horizon. While the predecessors of `clingo 4` relied on a `volatile` directive to provide a rigid mechanism for the expiration of transient rules, `clingo 4` captures the respective functionalities and customizations thereof in terms of the directive `#external`. This directive goes back to `lparse` (Syrjänen) and was also supported by the predecessors of `clingo 4` to exempt (input) atoms from simplifications fixing them to false. The `#external` directive of `clingo 4` provides a generalization that, in particular, allows for a flexible handling of yet undefined atoms.

For continuously assembling ground rules evolving at different stages of a reasoning process, `#external` directives declare atoms that may still be defined by rules added later on. As detailed in (Gebser et al. 2014), such atoms correspond to inputs in terms of module theory (Oikarinen and Janhunen 2006), which (unlike undefined output atoms) must not be simplified by fixing their truth value to false. In order to facilitate the declaration of input atoms, `clingo 4` supports schematic `#external` directives that are instantiated along with the rules of their respective subprograms. To this end, a directive like

```
#external p(X,Y) : q(X,Z), r(Z,Y).
```

is treated similar to the (virtual) rule `p(X,Y) :- q(X,Z), r(Z,Y)` during grounding. However, the head atoms of resulting ground instances are merely collected as (external) inputs, whereas the ground rules as such are discarded. Given this, a subprogram from the `clingo 4` input consists of all rules within the scope of `#program` directives with the same name and number of parameters, where `base` without parameters is used by default, along with virtual rules capturing `#external` directives in the same scope (see (Gebser et al. 2014) for details).

The instantiation of a subprogram `R` with a list `t_1, . . . , t_k` of terms, such as `acid(k)` above, relies on a list `t_1, . . . , t_k` of terms to replace occurrences of `c_1, . . . , c_k` with, both in original rules and virtual rules capturing `#external` directives in `R`. The parameter replacement yields a subprogram `R(c_1/t_1, . . . , c_k/t_k)`, which is instantiated relative to inputs. For instance, providing the term `42` for parameter `k` leads to `acid(k/42)` consisting of the fact `b(42)`. Control instructions guide the instantiation and assembly of subprograms, where `ground` instructions issued before the first or in-between two `solve` commands determine rules to instantiate and join with a module representing the previous state of `clingo 4`.

To sum up, schematic `#external` directives are embedded into the grounding process for a convenient declaration of input atoms from other subprogram instances. Given that they do not contribute ground rules, but merely qualify (undefined) atoms that should be exempted from simplifications, `#external` directives address the signature of subprograms’ ground instances. Hence, it is advisable to condition them by domain predicates\(^1\) (Syrjänen) only, as this precludes any interferences between signatures and grounder implementations. As long as input atoms remain undefined, their truth values can be freely picked and modified in-between `solve`

\(^1\) Domain and built-in predicates have unique extensions that can be evaluated entirely by means of grounding.
#script(python)

from gringo import Fun, SolveResult

def init(val, default):
    return val if val != None else default

def main(prg):
    stop = str(init(prg.getConst("istop"), "SAT"))
    step = int(init(prg.getConst("iinit"), 0))

    prg.ground("base", [])
    while True:
        step += 1
        prg.ground("cumulative", [step])
        prg.assignExternal(Fun("query", [step]), True)
        print 'STEP {0}'.format(step)
        ret = prg.solve()
        if (stop == "SAT" and ret == SolveResult.SAT) or 
           (stop == "UNSAT" and ret == SolveResult.UNSAT):
            break
    prg.releaseExternal(Fun("query", [step]))

#end.

Listing 1: Python script implementing iclingo functionality in clingo (iclingo.lp)

commands via assignExternal instructions, which thus allow for configuring the inputs to modules representing clingo 4 states in order to select among their stable models. Unlike that, the predecessors iclingo and oclingo of clingo 4 always assigned input atoms to false, so that the addition of rules was necessary to accomplish switching truth values. However, for a well-defined semantics, clingo 4 like its predecessors builds on the assumption that the modules induced by subprograms’ instantiations are compositional, which essentially requires definitions of (head) atoms and mutual positive dependencies to be local to evolving ground programs (cf. (Gebser et al. 2008)).

3 Using clingo 4 in practice

As mentioned above, clingo 4 fully supersedes its special-purpose predecessors iclingo and oclingo. To illustrate this, we give in Listing 1 a slightly simplified version of iclingo’s control loop in Python. The full control loop (included in the release) mainly adds handling of further iclingo options. Roughly speaking, iclingo offers a step-oriented, incremental approach to ASP that avoids redundancies by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem (as in iterative deepening search). To this end, a program is partitioned into a base part, describing static knowledge independent of the step parameter \( t \), a cumulative part, capturing knowledge accumulating with increasing \( t \), and a volatile part specific for each value of \( t \). These parts were delineated in iclingo by the directives \#base, \#cumulative \( t \), and \#volatile \( t \). In clingo 4, all three directives are captured by \#program declarations along with \#external for volatile rules.

We illustrate this by adapting the Towers of Hanoi encoding from (Gebser et al. 2012) in Figure 1. The problem instance in Figure 1(a) as well as Line 2 in 1(b) constitute static knowledge and thus belong to the base part. The transition function is described in the cumulative part in
Line 5–15 of Figure 1(b). Finally, the query is expressed in Line 18; its volatility is realized by making the actual goal condition `goal_on(D,P), not on(D,P,t)` subject to the truth assignment to the external atom `query(t)`. Grounding and solving of the program in Figure 1(a) and 1(b) is controlled by the Python script in Listing 1. Line 4–9 fix the stop criterion and initial value of the `step` variable. Both can be supplied as constants `istop` and `init` when invoking `clingo` 4. Once the base part is grounded in Line 11, the script loops until the stop criterion is met in Line 18–19. In each iteration, the current value of `step` is used in Line 14 and 15 to instantiate the subprogram `cumulative(t)` and to set the respective external atom `query(t)` to true. If the stop condition is yet unfulfilled w.r.t. the result of solving the extended program, the current `query(t)` atom is permanently falsified (cf. Line 17–20), thus annulling the corresponding instances of the integrity constraint in Line 18 of Figure 1(b) before they are replaced in the next iteration.

Another innovative feature of `clingo` 4 is its incremental optimization. This allows for adapting objective functions along the evolution of a program at hand. A simple example is the search for shortest plans when increasing the horizon in non-consecutive steps. To see this, recall that literals in minimize statements (and analogously weak constraints) are supplied with a sequence of terms of the form `w@p,⃗t`, where `w` and `p` are integers providing a weight and a priority level and `⃗t` is a sequence of terms (cf. (Calimeri et al. 2012)). As an example, consider the subprogram:

```prolog
#program cumulativeObjective(t).
#minimize { W@P,X,Y : move(X,Y,W,P,t) }.
\$ or :- move(X,Y,W,P,t). [W@P,X,Y,t]
```

When grounding and solving `cumulativeObjective(t)` for successive values of `t`, the solver’s objective function (per priority level `P`) is gradually extended with new atoms over `move/5`, and all previous ones are kept.

Moreover, for enabling the removal of literals from objective functions, we can use externals:
#program volatileObjective(t).
#external activateObjective(t).
#minimize{ W@P,X,Y,t : move(X,Y,W,P,t), activateObjective(t) }.

The subprogram volatileObjective(t) behaves like cumulativeObjective(t) as long as the external atom activateObjective(t) is true. Once it is set to false, all atoms over move/5 with the corresponding term for t are dismissed from objective functions.

A reasoning process in clingo 4 is partitioned into a sequence of solver invocations. We have seen how easily the solver’s logic program can be altered at each step. Sometimes it is useful to do this in view of a previously obtained stable model. For this purpose, the solve command can be equipped with an (optional) callback function onModel. For each stable model found during a call to solve(onModel), an object encompassing the model is passed to onModel, whose implementation can then access and inspect the model. A typical example is the addition of constraints based on the last model that are then supplied to the solver before computing the next one. An application is theory solving by passing (parts of) the last model to a theory solver for theory-based consistency checking or for providing the value of an externally evaluated objective function. Moreover, clingo 4 also furnishes an asynchronous solving function asolve that launches an interruptable solving process in the background. This is particularly useful in reactive settings in order to stop solving upon the arrival of new external information.

Similarly, the configuration of clasp can be changed at each step via the function setConf, taking a string including command line options along with a flag indicating whether the previous configuration is updated or replaced as arguments. For instance, this allows for changing search parameters, reasoning modes, number of threads, etc. Changing search parameters is of interest when addressing computational tasks involving the generation of several models, like optimal planning, multi-criteria optimization, or heuristic search. Apart from analyzing the previous model via the onModel callback, one can also monitor the search progress by means of the function getStats, returning an object encapsulating up to 135 attributes of the previous search process. Furthermore, clingo 4 allows for customizing the heuristic values of variables, as described in (Gebser et al. 2013a). At a higher level, a user may simply want to explore the set of models, and decide to compute first one, then all, and then the intersection or union of all models. This can be interleaved with the addition of subprograms via the function add, which may in turn include #external directives to declare temporary hypotheses. The experienced reader may note that this can be done fully interactively by means of IPython. Practical examples for the mentioned features can be found in the releases at (potassco).

4 Related work

Although clingo 3 (Gebser et al. 2011c) already featured Lua as an embedded scripting language, its usage was limited to (deterministic) computations during grounding; neither were library functions furnished by clingo 3.

Of particular interest is dlvhex (Fink et al. 2013), an ASP system aiming at the integration of external computation sources. For this purpose, dlvhex relies on higher-order logic programs using external higher-order atoms for software interoperability. Such external atoms should not be confused with clingo’s #external directive because they are evaluated via procedural means during solving. Given this, dlvhex can be seen as an ASP modulo Theory solver, similar to SAT modulo Theory solvers (Nieuwenhuis et al. 2006). In fact, dlvhex uses gringo and clasp as backends and follows the design of the ASP modulo CSP solver clingcon (Ostrowski and Schaub
In communicating with external “oracles” through clasp’s post propagation mechanism. In this way, theory solvers are tightly integrated into the ASP system and have access to the solver’s partial assignments. Unlike this, the light-weighted theory solving approach offered by clingo 4 can only provide access to total (stable) assignments. It is thus interesting future work to investigate in how far dlvhex can benefit from lifting its current low-level integration into clasp to a higher level in combination with clingo 4. Clearly, the above considerations also apply to extensions of dlvhex, such as acthex (Fink et al. 2013). Furthermore, jdlv (Febbraro et al. 2012) encapsulates the dlv system to facilitate one-shot ASP solving in Java environments by providing means to generate and process logic programs, and to afterwards extract their stable models.

The procedural attachment to the idp system (De Pooter et al. 2013; De Cat et al. 2014) builds on interfaces to C++ and Lua. Like clingo 4, it allows for evaluating functions during grounding, calling the grounder and solver multiple times, inspecting solutions, and reacting to external input after search. The emphasis, however, lies on high-level control blending in with idp’s modeling language, while clingo 4 offers more fine-grained control over the grounding and solving process, particularly aiming at a flexible incremental assembly of programs from subprograms.

In SAT, incremental solver interfaces from low-level APIs are common practice. Pioneering work was done in minisat (Eén and Sörensson 2004), furnishing a C++ interface for solving under assumptions. In fact, the clasp library underlying clingo 4 builds upon this functionality to implement incremental search (see (Gebser et al. 2008)). Given that SAT deals with propositional formulas only, solvers and their APIs lack support for modeling languages and grounding. Unlike this, the SAT modulo Theory solver z3 (de Moura and Björner 2008) comes with a Python API that, similar to clingo 4, provides a library for controlling the solver as well as language bindings for constraint handling. In this way, Python can be used as a modeling language for z3.

5 Discussion

The new clingo 4 system complements ASP’s declarative input language by control capacities expressed by embedded scripting languages. This is accomplished within a single integrated ASP grounding and solving process in which a logic program may evolve over time. The addition, deletion, and replacement of programs is controlled procedurally by means of clingo’s dedicated library. The incentives for evolving a logic program are manifold and cannot be captured with the standard one-shot approach of ASP. Examples include unrolling a transition function, as in planning, interacting with an environment, as in assisted living, robotics, or stream reasoning, interacting with a user exploring a domain, theory solving, and advanced forms of search. Addressing these demands by embedded scripting languages provides us with a generic and transparent approach. Unlike this, previous systems, like iclingo and oclingo, had a dedicated purpose involving rigid control capacities buried in monolithic programs. Rather than that, the basic technology of clingo 4 allows us to instantiate subprograms in-between solver invocations in a fully customizable way. On the declarative side, the availability of program parameters and the embedding of #external directives into the grounding process provide great flexibility in modeling schematic subprograms. In addition, the possibility of assigning input atoms facilitates the implementation of applications such as query answering or sliding window reasoning, as truth values can now be switched without manipulating a logic program.

The semantic underpinnings of our framework in terms of module theory capture the dynamic combination of logic programs in a generic way. It is interesting future work to investigate how
dedicated change operations whose interest was so far mainly theoretic, like updating (Alferes et al. 2002) or forgetting (Zhang and Foo 2006), can be put into practice within this framework.

The input language of clingo 4 extends the ASP-Core-2 standard (Calimeri et al. 2012). Although we have presented clingo 4 for normal logic programs, we mention that it accepts (extended) disjunctive logic programs, processed via the multi-threaded solving approach described in (Gebser et al. 2013b). In version 4.3, clingo moreover embeds clasp 3, featuring domain-specific heuristics (Gebser et al. 2013a) and optimization using unsatisfiable cores (Andres et al. 2012). clingo 4 is freely available at (potassco), and its releases include many best practice examples illustrating the aforementioned application scenarios.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1.

References


Clingo = ASP + Control: Preliminary Report

Delgrande and Faber (2011), 54–66.


