CPLUS2ASP: Computing Action Language $\mathcal{C}+$ in Answer Set Programming

Joseph Babb and Joohyung Lee

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, USA
{Joseph.Babb,joolee}@asu.edu

Abstract. We present Version 2 of system CPLUS2ASP, which implements the definite fragment of action language $\mathcal{C}+$. Its input language is fully compatible with the language of the Causal Calculator Version 2, but the new system is significantly faster thanks to modern answer set solving techniques. The translation implemented in the system is a composition of several recent theoretical results. The system orchestrates a tool chain, consisting of F2LP, CLINGO, ICLINGO, and AS2TRANSITION. Under the incremental execution mode, the system translates a $\mathcal{C}+$ description into the input language of ICLINGO, exploiting its incremental grounding mechanism. The correctness of this execution is justified by the module theorem extended to programs with nested expressions. In addition, the input language of the system has many useful features, such as external atoms by means of Lua calls and the user interactive mode. The system supports extensible multi-modal translations for other action languages, such as $\mathcal{B}$ and $\mathcal{BC}$, as well.

1 Introduction

Action language $\mathcal{C}+$ is a high level language for nonmonotonic causal theories, which allows us to describe transition systems succinctly [1]. The definite fragment of $\mathcal{C}+$ is expressive enough to represent various properties of actions, and was implemented in Version 2 of the Causal Calculator (CCALC)[1]. The system translates an action description in $\mathcal{C}+$ into formulas in propositional logic and calls SAT solvers to compute the models. Though CCALC is not a highly optimized system, it has been used to solve several challenging commonsense reasoning problems, including problems of nontrivial size [2], to provide a group of robots with high-level reasoning [3], to give executable specifications of norm-governed computational societies [4, 5], and to automate the analyses of business processes under authorization constraints [6].

An alternative way to compute the definite fragment of Boolean-valued $\mathcal{C}+$ is to translate it into answer set programs as studied in [7, 8]. The system reported in [9] and system COALA [10] are implementations of this method and accept the definite fragment of $\mathcal{C}$, a predecessor of language $\mathcal{C}+$. In particular, COALA was

[1] http://www.cs.utexas.edu/users/tag/cc

shown to be effective for several benchmark problems due to efficiency of ASP solvers.

However, the input language of COALA is missing several important features of $\mathcal{C}+$, such as multi-valued fluents, defined fluents, additive fluents, defeasible causal laws, and syntactically complex formulas. Also, it does not support many useful language constructs allowed in the input language of CCALC, such as user-defined macros, implicit declarations of sorts, and external atoms.

The design aim of system CPLUS2ASP [11] is to utilize the efficient ASP solving techniques as in COALA while supporting the full features of the input language of CCALC. Its design utilizes a standard library with meta-level sorts and meta-level variables, which yields a simple modular and extensible method to represent CCALC input programs in ASP. However, the first version of the system was a prototype implementation for a proof of concept.

This paper presents Version 2 of CPLUS2ASP, which is significantly enhanced in several ways.

- Its input language is fully compatible with the language of CCALC incorporating the features that were missing in CPLUS2ASP v1.
- The system supports extensible multi-modal translations for different action languages. Currently, in addition to $\mathcal{C}+$, the system supports language $\mathcal{B}$ [12], and a recently proposed language $\mathcal{BC}$ [13]. Language $\mathcal{BC}$ combines features of languages $\mathcal{B}$ and $\mathcal{C}$, and allows Prolog-style recursive definitions, which are not allowed in $\mathcal{C}+$.
- The system provides two execution modes: the command line mode and the interactive mode. The interactive mode gives a user-friendly interface for running various commands.
- In CCALC, external atoms are useful for some deterministic computation which is difficult to express directly in $\mathcal{C}+$. For example, they were utilized in [3] for a loose integration of task planning and motion planning. The new version of CPLUS2ASP supports this feature by utilizing Lua call available in the language of GRINGO.
- The new system provides an incremental computation of action descriptions, which often saves a significant amount of time. Since the translation of action descriptions into answer set programs may contain complex formulas, the justification of this computation uses the module theorem from [14], which extends the module theorem from [15] to first-order formulas under the stable model semantics [16].

In [11], the translation of a definite $\mathcal{C}+$ description into the input language of ASP solvers was explained in multiple steps. A $\mathcal{C}+$ description is first turned into a multi-valued causal theory, and then to a Boolean-valued causal theory by the method described in [17]. The resulting theory is further turned into logic programs with nested expressions by the translation in [8], and then the translation in [18] is applied to turn it into the input language of GRINGO.

In Section 2, we explain the translation in a simpler way by avoiding reference to causal theories but instead by using a recent proposal of multi-valued

propositional formulas under the stable model semantics [19]. A $\mathcal{C}+$ description is turned into multi-valued formulas under the stable model semantics, which is further turned into propositional formulas under the stable model semantics [20]. The result is further turned into the input language of GRINGO by the translation described in [18]. Section 3 introduces system CPLUS2ASP v2 and the features of its input language, and Section 4 compares the system with other similar systems. Our experiments show that the new system is significantly faster than the others.

2 From $\mathcal{C}+$ to ASP

2.1 Review: Multi-Valued Propositional Formulas

A *(multi-valued propositional) signature* is a set σ of symbols called *constants*, along with a nonempty finite set $Dom(c)$ of symbols, disjoint from σ, assigned to each constant c. $Dom(c)$ is called the *domain* of c. A *Boolean* constant is one whose domain is the set $\{\text{TRUE}, \text{FALSE}\}$. An *atom* of a signature σ is an expression of the form $c{=}v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *(multi-valued propositional) formula* of σ is a propositional combination of atoms. We often write $G \leftarrow F$, in a rule form as in logic programs, to denote the implication $F \rightarrow G$. A finite set of formulas is identified with the conjunction of the formulas in the set.

A *(multi-valued propositional) interpretation* of σ is a function that maps every element of σ to an element in its domain. An interpretation I *satisfies* an atom $c{=}v$, (symbolically, $I \models c{=}v$) if $I(c) = v$. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives. I is a *model* of a formula if I satisfies it. We often write an interpretation I with the set of atoms $c{=}v$ such that $I(c) = v$.

The *stable* models of a multi-valued propositional formula can be defined in terms of a reduct [19]. Let F be a multi-valued propositional formula of signature σ, and let I be a multi-valued propositional interpretation of σ. The reduct F^I of a multi-valued propositional formula F relative to a multi-valued propositional interpretation I is the formula obtained from F by replacing each maximal subformula that is not satisfied by I with $\bot$. I is a *(multi-valued) stable model* of F if I is the unique multi-valued interpretation of σ that satisfies F^I.

Example 1. Assume $\sigma = \{c\}$, and $Dom(c) = \{1, 2, 3\}$. Each of the three interpretations is a model of $c{=}1 \leftarrow c{=}1$, but none of them is stable because each reduct has no unique model. Formula $c{=}1 \leftarrow \neg\neg(c{=}1)$ has the same models as $c{=}1 \leftarrow c{=}1$, but it has one stable model, $\{c{=}1\}$: the reduct of the formula relative to this interpretation is $c{=}1 \leftarrow \neg\bot$, and $\{c{=}1\}$ is its unique model. Similarly, one can check that $(c{=}1 \leftarrow \neg\neg(c{=}1)) \wedge (c{=}2)$ has only one stable model $\{c{=}2\}$, which illustrates nonmonotonicity of the semantics.

2.2 $\mathcal{C}+$ as Multi-valued Propositional Formulas under SM

Begin with a multi-valued signature partitioned into *fluent* constants and *action* constants. The fluent constants are assumed to be further partitioned into *simple* and *statically determined*.

A *fluent formula* is a formula such that all constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent constants.

A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{1}$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the form (1) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{2}$$

where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants. A *causal law* is a static law, or an action dynamic law, or a fluent dynamic law. An *action description* is a finite set of causal laws.

An action description is called *definite* if F in every causal law (1) and (2) is either an atom or $\bot$.

For any definite action description D and any nonnegative integer m, the multi-valued propositional theory *cplus2mvpf*(D, m) ("$\mathcal{C}+$ to multi-valued propositional formulas") is defined as follows.[2] The signature of *cplus2mvpf*(D, m) consists of the pairs $i\!:\!c$ such that

- $i \in \{0, \dots, m\}$ and c is a fluent constant of D, or
- $i \in \{0, \dots, m-1\}$ and c is an action constant of D.

The domain of $i : c$ is the same as the domain of c. Recall that by $i : F$ we denote the result of inserting $i\!:$ in front of every occurrence of every constant in a formula F, and similarly for a set of formulas. The rules of *cplus2mvpf*(D, m) are:

$$i\!:\!F \leftarrow \neg\neg(i\!:\!G) \tag{3}$$

for every static law (1) in D and every $i \in \{0, \dots, m\}$, and for every action dynamic law (1) in D and every $i \in \{0, \dots, m-1\}$;

$$i\!:\!F \leftarrow \neg\neg(i\!:\!G) \wedge (i{-}1\!:\!H) \tag{4}$$

for every fluent dynamic law (2) in D and every $i \in \{1, \dots, m\}$;

$$0\!:\!c{=}v \leftarrow \neg\neg(0\!:\!c{=}v) \tag{5}$$

[2] The translation can be applied to non-definite $\mathcal{C}+$ descriptions as well, but then the semantics does not agree with $\mathcal{C}+$.

for every simple fluent constant c and every $v \in Dom(c)$.

Note how the definition of $cplus2mvpf(D, m)$ treats simple fluent constants and statically determined fluent constants in different ways: rules (5) are included only when c is simple.

The translation of $\mathcal{BC}$ into multi-valued propositional formulas is similar. Due to lack of space, we refer the reader to [13, Section 9].

2.3 Translating Multi-Valued Propositional Formulas to Propositional Formulas under SM

Note that even when we restrict attention to Boolean constants only, the stable model semantics for multi-valued propositional formulas does not coincide with the stable model semantics for propositional formulas. Syntactically, they are different (one uses expressions of the form $c = \text{TRUE}$ and $c = \text{FALSE}$; the other uses propositional atoms). Semantically, the former relies on the uniqueness of (Boolean)-functions, while the latter relies on the minimization on propositional atoms. Nonetheless there is a simple reduction from the former to the latter.

Begin with a multi-valued propositional signature σ. By σ^{prop} we denote the signature consisting of Boolean constants $c(v)$ for all constants c in σ and all $v \in Dom(c)$. For any multi-valued propositional formula F of σ, by F^{prop} we denote the propositional formula that is obtained from F by replacing each occurrence of a multi-valued atom $c{=}v$ with $c(v)$. For any constant c with $Dom(c)$, by $UEC(c)$ we denote the existence and uniqueness constraints for c:

$$\bot \leftarrow (c(v) \wedge c(v'))$$

for all $v, v' \in Dom(c)$ such that $v \neq v'$, and

$$\bot \leftarrow \neg \bigvee_{v \in Dom(c)} c(v) \ .$$

By UEC_σ we denote the conjunction of $UEC(c)$ for all $c \in \sigma$.

For any interpretation I of σ, by I^{prop} we denote the interpretation of σ^{prop} that is obtained from I by defining $I^{prop} \models c(v)$ iff $I \models c{=}v$.

There is a one-to-one correspondence between the stable models of F and the stable models of F^{prop}. The following theorem is a special case of Corollary 1 from [19].

Theorem 1 *Let F be a multi-valued propositional formula of a signature σ such that, for every constant c in σ, $Dom(c)$ has at least two elements.* **(I)** *An interpretation I of σ is a multi-valued stable model of F iff I^{prop} is a propositional stable model of $F^{prop} \wedge UEC_\sigma$.* **(II)** *An interpretation J of σ^{prop} is a propositional stable model of $F^{prop} \wedge UEC_\sigma$ iff $J = I^{prop}$ for some multi-valued stable model I of F.*

2.4 Incremental Computation of $\mathcal{C}+$

In answer set planning [21], the length of a plan needs to be specified. When the length is not known in advance, a plan can be found by iteratively increasing the

possible plan length. CPLUS2ASP Version 1 calls CLINGO for each such iteration, resulting in redundant computations each time.

Instead, by default, CPLUS2ASP v2 uses ICLINGO, which accepts *incremental logic programs*. Gebser *et al.* [22] define an incremental logic program to be a triple $\langle B, P[t], Q[t]\rangle$, where B is a disjunctive logic program, and $P[t]$, $Q[t]$ are incrementally parameterized disjunctive logic programs. Informally, B is the *base* program component, which describes static knowledge; $P[t]$ is the *cumulative* program component, which contains information regarding every step t that should be accumulated during execution; $Q[t]$ is the *volatile query* program component, containing constraints or information regarding the final step. Conceptually, system ICLINGO computes $B \cup P[1] \cup \cdots \cup P[k] \cup Q[k]$ by increasing k one by one, but avoids reproducing ground rules in each step. Also, previously learned heuristics, conflicts, or loops are reused without having to recompute them. This method turns out to be quite effective. The correctness of this computation assumes that $\langle B, P[t], Q[t]\rangle$ is *acyclic* [14].

Below we show that the translation from $\mathcal{C}+$ described previously can be modified to yield an incremental logic program, which is always acyclic, and thus can be computed by ICLINGO.

For any $\mathcal{C}+$ description D, and any formula $F(t)$ (called a *query*) of the same signature as *cplus2mvpf*(D, t), where t is a parameter denoting a nonnegative integer, we define the corresponding incremental logic program $\langle B, P[t], Q[t]\rangle$ as follows:

- B consists of
  - $0\!:\!UEC(f)$ for every fluent constant f;
  - $0\!:\!c(v) \leftarrow \neg\neg(0\!:\!c(v))$ for every simple fluent c and every $v \in Dom(c)$;
  - $0\!:\!F^{prop} \leftarrow \neg\neg(0\!:\!G^{prop})$ for every static law (1) in D.
- $P[t]$ $(t \geq 1)$ consists of
  - $t\!:\!UEC(f)$ for every fluent constant f;
  - $(t{-}1)\!:\!UEC(a)$ for every action constant a;
  - $t\!:\!F^{prop} \leftarrow \neg\neg(t\!:\!G^{prop})$ for every static law (1) in D;
  - $(t{-}1)\!:\!F^{prop} \leftarrow \neg\neg((t{-}1)\!:\!G^{prop})$ for every action dynamic law (1) in D;
  - $t\!:\!F^{prop} \leftarrow \neg\neg(t\!:\!G^{prop}) \wedge ((t{-}1)\!:\!H^{prop})$ for every fluent dynamic law (2) in D.
- $Q[t]$ is $\bot \leftarrow \neg(F[t])^{prop}$.

Upon receiving this input and a range of nonnegative integers $[\min \ldots \max]$, ICLINGO will find an answer set of the module $\mathbf{R}_k$ with $k = \min, \min + 1, \ldots$ until it finds an answer set, or $k = \max$, whichever comes first. In [14], module $\mathbf{R}_k$ is defined from $\langle B, P[t], Q[t]\rangle$ as follows.

$$\begin{aligned}
\mathbf{P}_0 &= FM(B, \emptyset), \\
\mathbf{P}_i &= \mathbf{P}_{i-1} \sqcup FM(P[i], Out(\mathbf{P}_{i-1})), \qquad (1 \leq i \leq k) \\
\mathbf{R}_k &= \mathbf{P}_k \sqcup FM(Q[k], Out(\mathbf{P}_k)) \ .
\end{aligned}$$

(Due to lack of space, we refer the reader to [14] for the notations.)

The following theorem states the correctness of incremental execution in CPLUS2ASP.

Theorem 2 *For any definite $\mathcal{C}+$ description D, any non-negative integer k, and any formula $F(k)$ of the same signature as $cplus2mvpf(D,k)$, an interpretation I is a multi-valued stable model of $cplus2mvpf(D,k) \cup \{\bot \leftarrow \neg F(k)\}$ iff I^{prop} is a stable model of $\mathbf{R}_k$. Conversely, an interpretation J is a stable model of $\mathbf{R}_k$ iff $J = I^{prop}$ for some multi-valued stable model of $cplus2mvpf(D,k) \cup \{\bot \leftarrow \neg F(k)\}$.*

Proof. (Sketch) We can check that $\langle B, P[t], Q[t]\rangle$ obtained from the $\mathcal{C}+$ description and a query as above is acyclic according to Definition 12 from [14]. Then the claim follows from Proposition 5 from [14]. ∎

The translation of $\mathcal{BC}$ into an incremental logic program is similar.

3 System Cplus2ASP v2

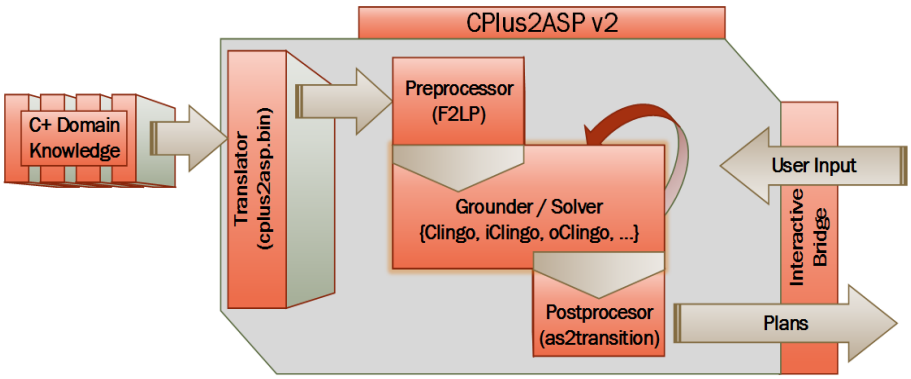

Fig. 1. CPLUS2ASP v2 System Architecture

System CPLUS2ASP v2 is a re-engineering of the prototypical CPLUS2ASP v1 system [11] and is available under Version 3 of the GNU Public License. Like its predecessor, CPLUS2ASP v2 uses a highly modular architecture that is designed to take advantage of the existing tools, including system F2LP [18] and highly-optimized ASP grounders and solvers in addition to a number of packaged sub-components. Figure 1 shows a high-level conceptualization of the interaction of the sub-components in the CPLUS2ASP v2 architecture.

For a description of the input language of CPLUS2ASP, we refer the reader to the CPLUS2ASP homepage at `http://reasoning.eas.asu.edu/cplus2asp` or CCALC 2 homepage at `http://www.cs.utexas.edu/~tag/ccalc/`. A typical run of CPLUS2ASP involves the user interacting with the interactive bridge, a tightly-coupled shell-like interface for CPLUS2ASP, in order to configure the CPLUS2ASP run. CPLUS2ASP.BIN, a translator sub-component, is then called to compile a CCALC 2 input program into a logic program containing complex formulas. Following this, system F2LP further turns the program in the input language of GRINGO. The result of this compilation is given to CLINGO, or

a similar answer set solver, and one or more answer sets are calculated. Finally, AS2TRANSITION is invoked in order to format the answer sets into a readable format.

CPLUS2ASP accepts a CCALC 2 style syntax of language $\mathcal{BC}$ as well, for which the user can select a different language mode for running. In addition, CPLUS2ASP is able to provide two target translations, a *static* translation to traditional ASP, and an *incremental* variant, as described in section 2.4.

3.1 Running Modes of System Cplus2ASP v2

In this section we briefly review the usage of CPLUS2ASP v2. For more complete documentation and information on obtaining and installing CPLUS2ASP v2 we invite the reader to visit the CPLUS2ASP homepage.

CPLUS2ASP v2 currently offers two distinct user-interaction methods: command-line and interactive shell. A brief introduction to both modes is provided below.

Using the Command-Line Mode The command-line mode is designed primarily for interacting with a script or a seasoned CPLUS2ASP user who is familiar with the options available to them. The command-line mode is the default user-interaction mode when a query is provided while calling CPLUS2ASP.

For instance, to run a query labeled "`simple`" on a $\mathcal{C}+$ description stored in file `bw-test`, one can run the command:

```
cplus2asp bw-test query=simple
```

In order to run the command under the $\mathcal{BC}$ semantics, the flag `--language=bc` should be asserted in the command line call.[3]

If more solutions are desired, the number of solutions can be appended to the end of the command-line. As an example, appending `4` to the end of the command will return up to four solutions, while appending `all` or `0` will return all solutions.

The system provides the following options to write the output of a toolchain component into a file. Below `[PROGRAM]` may be one of `pre-processor`, `grounder`, `solver`, or `post-processor`.

`--[PROGRAM]-output=[FILE]` Writes the output of the toolchain component `[PROGRAM]` to a persistent output file `[FILE]`.

`--to-[PROGRAM]` Executes the program toolchain up to and including `[PROGRAM]`. Similarly, `--from-[PROGRAM]` selects a program to initiate execution with and continue from.

As an example, if the user wants to run the toolchain up to the preprocessor and store the results for use later, he could use the command

```
cplus2asp bw-test --to-pre-processor > bw-test.lp.
```

[3] The `bw-test` example program, along with other examples, can be found from the CPLUS2ASP homepage.

Later, he could then run the command

```
cplus2asp bw-test.lp --from-grounder query=simple
```

to continue execution.

Using the Interactive Mode The user-interactive mode provides a shell-like interface which allows the user to perform many of the configurations available from the command line. In general, the user-interactive mode is entered any time the user fails to provide all necessary information within the command-line arguments. As such, the easiest way to enter the user-interactive mode is to neglect to specify a query on the command-line. As an example, the command

```
cplus2asp bw-test
```

will enter the user-interactive mode.

While in the user-interactive mode, the following commands, among others, are available to the user:

`help` Displays the list of available commands.
`config` Reveals the currently selected running options.
`queries` Displays the list of available queries to run.
`minstep=[#]` Overrides the minimum step to solve for the next query selected.
`maxstep=[#]` Overrides the maximum step to solve for the next query selected.
`sol=[#]` Selects the number of solutions to display.
`query=[QUERY]` Runs the selected query and returns the results.
`exit` Exits the program.

Following successful execution of a query, the system will return to the interactive prompt and the process can be repeated. For more information on using CPLUS2ASP v2, we invite the reader to explore the documentation available at `http://reasoning.eas.asu.edu/cplus2asp` or within the help usage message available by executing `cplus2asp --help`.

3.2 Lua in System Cplus2ASP v2

System CPLUS2ASP v2 allows for embedding external Lua function calls in the system, which are evaluated at grounding time. These Lua calls allow the user a great deal of flexibility when designing a program and can be used for complex computation that is not easily expressible in logic programs. A Lua function must be encapsulated in `#begin_lua ...#end_lua.` tags, and, can optionally be included in a separate file ending in `.lua`. Lua calls occurring within the CPLUS2ASP program are restricted to occurring within the `where` clause [4] of each rule and must be prefaced with an `@` sign.

For example, one can say moving a block does not always work.[5]

[4] The condition in the `where` clause is evaluated at grounding time.
[5] Note that this is decided at grounding time so this is not truly random.

```
move(B,L) causes loc(B)=L where @roll(1,2).
```

with Lua function defined as

```
#begin_lua
math.randomseed(os.time())
function roll(a,n)--returns 1 with probability a/n
  if(math.random(n) <= a) then return 1
  else return 0
  end
end
#end_lua.
```

A more complete description of the system's Lua functionality and additional examples of its use are available from the CPLUS2ASP homepage.

4 Experiments

In order to compare the performance of the CPLUS2ASP v2 system with its predecessors, we used large variants of several widely known domains [6] and compared the performance of CPLUS2ASP's running modes with the performance of CCALC v2, CPLUS2ASP v1, and the incremental and static running modes of COALA (where applicable). All experiments were performed on an Intel Core 2 Duo 3.00 GHZ CPU with 4 GB RAM running Ubuntu 11.10. The CCALC v2 tests used RELSAT 2.0 as a SAT solver while CPLUS2ASP v1, v2, and COALA tests used the same version of CLINGO, v3.0.5.

The domains tested include a large variant of the Traffic World [2], which models the behavior of cars on a road; a variant of the Blocks World where actions have costs [23]; the Spacecraft Integer [23], which models a spacecraft's movement with multiple independent jets; the Towers of Hanoi; and the Ferryman domain, which involves moving a number of wolves and sheep across a river without allowing the sheep to be eaten. The Towers of Hanoi and Ferryman descriptions are from examples packaged with COALA v1.0.1. In order to run on other systems, we manually turned them into the syntax of CCalc input language.

Table 1 compares the results of the test benchmarks for each of the available configurations. Each measured time includes translation, grounding, and solving for all possible maximum steps between 0 and the horizon (#), as well as the number of atoms and rules produced below each timing. In all test cases CPLUS2ASP's incremental running mode showed a significant performance advantage compared to the other systems, performing roughly 3 times faster than COALA's incremental mode and an order of magnitude faster than its predecessor CPLUS2ASP v1. COALA's incremental running mode comes in the second place in all but one benchmark. CPLUS2ASP v2's static mode tended to outperform its predecessor on the more computation-heavy domains with additive fluents,

[6] All benchmark programs are available from the CPLUS2ASP homepage.

Domain	**steps**	CCALC 2	CPLUS2ASP V1	COALA static	COALA incr.	CPLUS2ASP V2 static	CPLUS2ASP V2 incr.
traffic (altmerge)	11	878.59 s + 1 s [a] [531552 / 3671940]	95.43 s + 25.95 s [2722247 / 3341068]	_ [b] –	– –	82.16 s + 26.57 s [2262231 / 2766459]	14.2 s + 2.6 s
bw-cost (15) [c]	8	131.1 s + 5 s [149032 / 624439]	76.16 s + 0.4 s [123517 / 260282]	–	–	17.09 s + 3.16 s [43052 / 526923]	3.47 s + 0.16 s
bw-cost (20)	9	52 s + 987 s [374785 / 1584778]	271 s + 9.17 s [279869 / 626496]	–	–	63.26 s + 66.58 s [102426 / 1745166]	13.45 s + 2.24 s
spacecraft (15/8) [d]	3	173.62 s + 0 s [128262 / 622158]	16.07 s + 2.65 s [146056 / 146056]	–	–	5.57 s + 0.06 s [132918 / 253514]	2.33 s + 0.01 s
spacecraft (25/10)	4	*timeout*	208.2 s + 480.24 s [760673 / 1653650]	–	–	67.55 s + 3.42 s [732860 / 1427771]	17.46 s + 0.35 s
hanoi (6/3) [e]	64	14 s + 1983 s [13710 / 221895]	38.9 s + 137.27 s [37297 / 298047]	1039.15 s + 507.12 s [13798 / 410559]	1.4 s + 51.13 s	547.9 s + 47.53 s [10086 / 202694]	0.76 s + 3.5 s
towers (8/4)	33	*timeout*	31.19 s + 102.69 s [35041 / 433660]	304.02 s + 3017.87 s [12922 / 655436]	1.51 s + 470.23 s	102.81 s + 89.36 s [9074 / 324668]	1.04 s + 14.8 s
ferryman (10/4) [f]	16	39.45 s + 0 s [55905 / 308909]	8.27 s + 2.98 s [14122 / 120693]	40.85 s + 8.71 s [4973 / 358772]	0.87 s + 1.85 s	21.59 s + 2.37 s [12721 / 112912]	0.66 s + 0.25 s
ferryman (15/4)	26	1004.26 s + 0 s [256590 / 1452554]	85.21 s + 39.54 s [42687 / 539513]	793.13 s + 169.18 s [15718 / 2275992]	6.13 s + 14.73 s	318.4 s + 34.4 s [39536 / 515167]	4.18 s + 2.97 s

[a] preprocessing time + solving time [# atoms / # rules]
[b] The input language is not expressive enough to represent the domain.
[c] maximum cost
[d] domain size ($15 \times 15 \times 15$) / goal position ($8 \times 8 \times 8$)
[e] # disks / # pegs
[f] # animals / boat capacity

Table 1. Benchmarking Results

but was subsequently outmatched in the others. Finally, CCALC 2 and COALA's static mode came in last (with CCALC performing slightly worse in most cases).

Figure 2 shows a more detailed analysis of the execution of the first 100 steps of solving an extreme variant of the ferryman domain consisting of 120 of each animal by graphing the time spent (in seconds) on each step by each configuration. While the static configurations were required to completely re-ground and re-solve the translated answer set program for each maximum step, resulting in an ever-growing amount of work to be performed at each step, CPLUS2ASP v2's incremental running mode is able to avoid this by only grounding the new cumulative ($P[t]$) and volatile ($Q[t]$) components and leveraging heuristics learned from previous iterations. This results in far less time being required for checking each increment.

Although COALA's incremental mode uses the same reasoning engine ICLINGO as CPLUS2ASP v2's incremental mode, system CPLUS2ASP sees a significant overall speed-up over COALA. This is related to a significant reduction in the

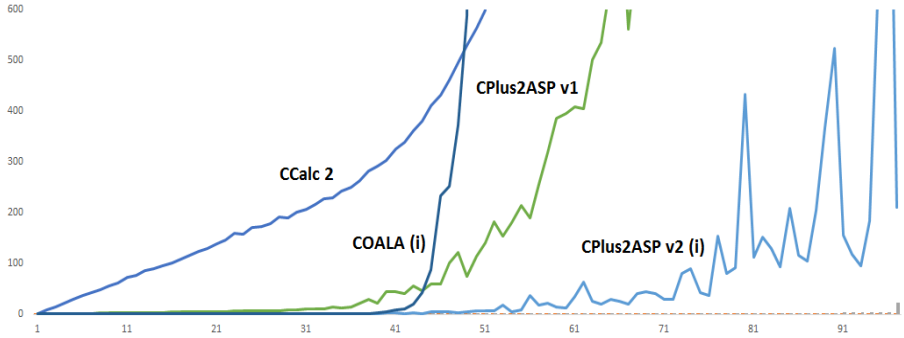

Fig. 2. Ferryman 120/4 Long Horizon Analysis

number of atoms and rules produced during grounding, which also accounts for far fewer conflicts and restarts during solving in all test cases.

5 Conclusion

A distinct advantage that CPLUS2ASP v2 has over its prototypical predecessor is that it was re-engineered in order to allow for far greater flexibility and extensibility via a multi-modal execution model. This makes it suitable for use as a base-platform for future input language implementations, input language extensions, or target languages/platforms.

The advances in ASP solving techniques account for the efficiency of system CPLUS2ASP. We expect that the significant speed-up of the system demonstrated by CPLUS2ASP v2, as well as the enhanced expressivity of the input language, will contribute to widening application of action languages in various domains.

Acknowledgements: We are grateful to Michael Bartholomew and the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-0916116 and by the South Korea IT R&D program MKE/KIAT 2010-TD-300404-001.

References

1. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153(1–2)** (2004) 49–104
2. Akman, V., Erdoğan, S., Lee, J., Lifschitz, V., Turner, H.: Representing the Zoo World and the Traffic World in the language of the Causal Calculator. Artificial Intelligence **153(1–2)** (2004) 105–140
3. Caldiran, O., Haspalamutgil, K., Ok, A., Palaz, C., Erdem, E., Patoglu, V.: Bridging the gap between high-level reasoning and low-level control. In: LPNMR. (2009) 342–354

4. Artikis, A., Sergot, M., Pitt, J.: Specifying norm-governed computational societies. ACM Transactions on Computational Logic **9**(1) (2009)
5. Desai, N., Chopra, A.K., Singh, M.P.: Representing and reasoning about commitments in business processes. In: AAAI. (2007) 1328–1333
6. Armando, A., Giunchiglia, E., Ponta, S.E.: Formal specification and automatic analysis of business processes under authorization constraints: an action-based approach. In: Proceedings of the 6th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'09). (2009)
7. McCain, N.: Causality in Commonsense Reasoning about Actions. PhD thesis, University of Texas at Austin (1997)
8. Ferraris, P., Lee, J., Lierler, Y., Lifschitz, V., Yang, F.: Representing first-order causal theories by logic programs. TPLP **12**(3) (2012) 383–412
9. Doğandağ, S., Alpaslan, F.N., Akman, V.: Using stable model semantics (SMODELS) in the Causal Calculator (CCALC). In: Proceedings 10th Turkish Symposium on Artificial Intelligence and Neural Networks. (2001) 312–321
10. Gebser, M., Grote, T., Schaub, T.: Coala: A compiler from action languages to ASP. In: Proceedings of European Conference on Logics in Artificial Intelligence (JELIA). (2010) 360–364
11. Casolary, M., Lee, J.: Representing the language of the causal calculator in answer set programming. In: ICLP (Technical Communications). (2011) 51–61
12. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence **3** (1998) 195–210
13. Lee, J., Lifschitz, V., Yang, F.: Action language BC: Preliminary report. In: In Proc. IJCAI 2013. (2013) To appear.
14. Babb, J., Lee, J.: Module theorem for the general theory of stable models. TPLP **12**(4-5) (2012) 719–735
15. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. Journal of Artificial Intelligence Research **35** (2009) 813–857
16. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence **175** (2011) 236–263
17. Lee, J.: Automated Reasoning about Actions. PhD thesis, University of Texas at Austin (2005)
18. Lee, J., Palla, R.: System F2LP – computing answer sets of first-order formulas. In: Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2009) 515–521
19. Bartholomew, M., Lee, J.: Stable models of formulas with intensional functions. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR). (2012) 2–12
20. Ferraris, P.: Answer sets for propositional theories. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2005) 119–131
21. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138** (2002) 39–54
22. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Springer (2011) 54–66
23. Lee, J., Lifschitz, V.: Describing additive fluents in action language $\mathcal{C}+$. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). (2003) 1079–1084