System ASPMT2SMT:
Computing ASPMT Theories by SMT Solvers

Michael Bartholomew and Joohyung Lee

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

Abstract. Answer Set Programming Modulo Theories (ASPMT) is an approach to combining answer set programming and satisfiability modulo theories based on the functional stable model semantics. It is shown that the tight fragment of ASPMT programs can be turned into SMT instances, thereby allowing SMT solvers to compute stable models of ASPMT programs. In this paper we present a compiler called ASPSMT2SMT, which implements this translation. The system uses ASP grounder GRINGO and SMT solver Z3. GRINGO partially grounds input programs while leaving some variables to be processed by Z3. We demonstrate that the system can effectively handle real number computations for reasoning about continuous changes.

1 Introduction

Answer Set Programming (ASP) is a widely used declarative computing paradigm. Its success is largely due to the expressivity of its modeling language and efficiency of ASP solvers thanks to intelligent grounding and efficient search methods that originated from propositional satisfiability (SAT) solvers. While grounding methods implemented in ASP solvers are highly optimized, ASP inherently suffers when variables range over large domains. Furthermore, real number computations are not supported by ASP solvers because grounding cannot be even applied. Thus reasoning about continuous changes even for a small interval requires loss of precision by discretizing the domain.

Satisfiability Modulo Theories (SMT) emerged as an enhancement of SAT, which can be also viewed as a special case of (decidable) first-order logic in which certain predicate and function symbols in background theories have fixed interpretations. Example background theories are the theory of real numbers, the theory of linear arithmetic, and difference logic.

A few approaches to loosely combining ASP and SMT/CSP exist [1–3], in which nonmonotonicity of the semantics is related to predicates in ASP but has nothing to do with functions in SMT/CSP. For instance, while

$$WaterLevel(t{+}1, tank, l) \;\leftarrow\; WaterLevel(t, tank, l),\; not\; \neg WaterLevel(t{+}1, tank, l)$$

(t is a variable ranging over steps; l is a variable for the water level) represents the default value of water level correctly (albeit grounding suffers when the variables range

over a large numeric domain), rewriting it in the language of CLINGCON—a combination of ASP solver CLINGO and constraint solver GECODE—as

$$\textit{WaterLevel}(t{+}1, \textit{tank}) =^{\$} l \;\leftarrow\; \textit{WaterLevel}(t, \textit{tank}) =^{\$} l, \; \textit{not} \; \neg(\textit{WaterLevel}(t{+}1, \textit{tank}) =^{\$} l)$$

does not express the concept of defaults correctly.

In [4], it was observed that a tight integration of ASP and SMT requires a generalization of the stable model semantics in which default reasoning can be expressed via (non-Herbrand) functions as well as predicates. Based on the functional stable model semantics from [5], a new framework called "Answer Set Programming Modulo Theories (ASPMT)" was proposed, which is analogous to SMT. Just like SMT is a generalization of SAT and, at the same time, a special case of first-order logic with fixed background theories, ASPMT is a generalization of the traditional ASP and, at the same time, a special case of the functional stable model semantics in which certain background theories are assumed. Unlike SMT, ASPMT allows expressive nonmonotonic reasoning as allowed in ASP.

It is shown in [4], a fragment of ASPMT instances can be turned into SMT instances, so that SMT solvers can be used for computing stable models of ASPMT instances. In this paper, we report an implementation of this translation in the system called "ASPMT2SMT." The system first partially grounds the theory by replacing "ASP variables" with ground terms, leaving other "SMT variables" ungrounded. Then, it computes the completion of the theory. Under certain conditions guaranteed by the class of ASPMT theories considered, the remaining variables can then be eliminated. After performing this elimination, the ASPMT2SMT system then invokes the Z3 system to compute classical models, which correspond to the stable models of the original theory. We show that several examples involving both discrete changes as well as continuous changes can be naturally represented in the input language of ASPMT2SMT, and can be effectively computed.

The paper is organized as follows. In section 2, we first review the functional stable models semantics and as its special case, ASPMT, and then review the theorem on completion from [5]. In section 3, we describe the process of variable elimination used by the system. In section 4, we describe the architecture of the system as well as the syntax of the input language. Finally, in section 5, we present several experiments with and without continuous reasoning and compare the performance to ASP solver CLINGO when appropriate.

The system is available at `http://reasoning.eas.asu.edu/aspmt`.

2 Preliminaries

2.1 Review of the Functional Stable Model Semantics

We review the stable model semantics of intensional functions from [5]. Formulas are built the same as in first-order logic.

Similar to circumscription, for predicate symbols (constants or variables) u and c, expression $u \leq c$ is defined as shorthand for $\forall \mathbf{x}(u(\mathbf{x}) \rightarrow c(\mathbf{x}))$. Expression $u = c$ is defined as $\forall \mathbf{x}(u(\mathbf{x}) \leftrightarrow c(\mathbf{x}))$ if u and c are predicate symbols, and $\forall \mathbf{x}(u(\mathbf{x}) = c(\mathbf{x}))$ if

they are function symbols. For lists of symbols $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{c} = (c_1, \ldots, c_n)$, expression $\mathbf{u} \leq \mathbf{c}$ is defined as $(u_1 \leq c_1) \wedge \cdots \wedge (u_n \leq c_n)$, and similarly, expression $\mathbf{u} = \mathbf{c}$ is defined as $(u_1 = c_1) \wedge \cdots \wedge (u_n = c_n)$. Let $\mathbf{c}$ be a list of distinct predicate and function constants, and let $\widehat{\mathbf{c}}$ be a list of distinct predicate and function variables corresponding to $\mathbf{c}$. By $\mathbf{c}^{pred}$ ($\mathbf{c}^{func}$, respectively) we mean the list of all predicate constants (function constants, respectively) in $\mathbf{c}$, and by $\widehat{\mathbf{c}}^{pred}$ ($\widehat{\mathbf{c}}^{func}$, respectively) the list of the corresponding predicate variables (function variables, respectively) in $\widehat{\mathbf{c}}$.

For any formula F and any list of predicate and function constants $\mathbf{c}$, which we call *intensional* constants, expression $\mathrm{SM}[F;\ \mathbf{c}]$ is defined as

$$F \wedge \neg \exists \widehat{\mathbf{c}}(\widehat{\mathbf{c}} < \mathbf{c} \wedge F^*(\widehat{\mathbf{c}})),$$

where $\widehat{\mathbf{c}} < \mathbf{c}$ is shorthand for $(\widehat{\mathbf{c}}^{pred} \leq \mathbf{c}^{pred}) \wedge \neg(\widehat{\mathbf{c}} = \mathbf{c})$, and $F^*(\widehat{\mathbf{c}})$ is defined recursively as follows.

- When F is an atomic formula, F^* is $F' \wedge F$ where F' is obtained from F by replacing all intensional (function and predicate) constants $\mathbf{c}$ in it with the corresponding (function and predicate) variables from $\widehat{\mathbf{c}}$;
- $(G \wedge H)^* = G^* \wedge H^*$; $(G \vee H)^* = G^* \vee H^*$;
- $(G \rightarrow H)^* = (G^* \rightarrow H^*) \wedge (G \rightarrow H)$;
- $(\forall x G)^* = \forall x G^*$; $(\exists x F)^* = \exists x F^*$.

(We understand $\neg F$ as shorthand for $F \rightarrow \bot$; $\top$ as $\neg\bot$; and $F \leftrightarrow G$ as $(F \rightarrow G) \wedge (G \rightarrow F)$.)

When F is a sentence, the models of $\mathrm{SM}[F; \mathbf{c}]$ are called the *stable* models of F *relative to* $\mathbf{c}$. They are the models of F that are "stable" on $\mathbf{c}$. The definition can be easily extended to formulas of many-sorted signatures.

This definition of a stable model is a proper generalization of the one from [6], which views logic programs as a special case of first-order formulas.

We will often write $G \leftarrow F$, in a rule form as in logic programs, to denote the universal closure of $F \rightarrow G$. A finite set of formulas is identified with the conjunction of the formulas in the set.

By $\{c = v\}$, we abbreviate the formula $c = v \vee \neg(c = v)$ which, in the functional stable model semantics, can be intuitively understood as "by default, c is mapped to v".

2.2   ASPMT as a Special Case of the Functional Stable Model Semantics

We review the semantics of ASPMT described in [4]. Formally, an SMT instance is a formula in many-sorted first-order logic, where some designated function and predicate constants are constrained by some fixed background interpretation. SMT is the problem of determining whether such a formula has a model that expands the background interpretation [7].

The syntax of ASPMT is the same as that of SMT. Let σ^{bg} be the (many-sorted) signature of the background theory bg. An interpretation of σ^{bg} is called a *background interpretation* if it satisfies the background theory. For instance, in the theory of reals, we assume that σ^{bg} contains the set $\mathcal{R}$ of symbols for all real numbers, the set of arithmetic functions over real numbers, and the set $\{<, >, \leq, \geq, =\}$ of binary predicates

over real numbers. Background interpretations interpret these symbols in the standard way.

Let σ be a signature that is disjoint from σ^{bg}. We refer to functions in σ^{bg} as interpreted functions and functions in σ as uninterpreted functions. We say that an interpretation I of σ satisfies F w.r.t. the background theory bg, denoted by $I \models_{bg} F$, if there is a background interpretation J of σ^{bg} that has the same universe as I, and $I \cup J$ satisfies F. For any ASPMT sentence F with background theory σ^{bg}, interpretation I is a *stable model* of F relative to $\mathbf{c}$ (w.r.t. background theory σ^{bg}) if $I \models_{bg} \mathrm{SM}[F; \mathbf{c}]$. When $\mathbf{c}$ is empty, the stable models of F coincides with the models of F.

Consider the following running example from a Texas Action Group discussion[1].

> A car is on a road of length L. If the accelerator is activated, the car will speed up with constant acceleration A until the accelerator is released or the car reaches its maximum speed MS, whichever comes first. If the brake is activated, the car will slow down with acceleration $\neg A$ until the brake is released or the car stops, whichever comes first. Otherwise, the speed of the car remains constant. Give a formal representation of this domain, and write a program that uses your representation to generate a plan satisfying the following conditions: at duration 0, the car is at rest at one end of the road; at duration T, it should be at rest at the other end.

This problem is an instance of planning with continuous time, which requires real number computations.

The domain can be naturally represented in ASPMT as follows. Below s ranges over time steps, b is a boolean variable, x, y, a, c, d are all real variables, and A and MS are some specific numbers.

We represent that the actions *Accel* and *Decel* are exogenous and the duration of each time step is to be arbitrarily selected as

$$\{Accel(s) = b\}, \qquad \{Decel(s) = b\}, \qquad \{Duration(s) = x\}.$$

Both *Accel* and *Decel* cannot be performed at the same time:

$$\bot \leftarrow Accel(s) = \mathrm{TRUE} \wedge Decel(s) = \mathrm{TRUE}.$$

The effects of *Accel* and *Decel* on *Speed* are described as

$$Speed(s+1) = y \leftarrow Accel(s) = \mathrm{TRUE} \ \wedge \ Speed(s) = x \ \wedge \ Duration(s) = d$$
$$\wedge \ (y = x + \mathrm{A} \times d),$$
$$Speed(s+1) = y \leftarrow Decel(s) = \mathrm{TRUE} \ \wedge \ Speed(s) = x \ \wedge \ Duration(s) = d$$
$$\wedge \ (y = x - \mathrm{A} \times d).$$

The preconditions of *Accel* and *Decel* are described as

$$\bot \leftarrow Accel(s) = \mathrm{TRUE} \ \wedge \ Speed(s) = x \ \wedge \ Duration(s) = d$$
$$\wedge \ (y = x + \mathrm{A} \times d) \ \wedge \ (y > \mathrm{MS}),$$
$$\bot \leftarrow Decel(s) = \mathrm{TRUE} \ \wedge \ Speed(s) = x \ \wedge \ Duration(s) = d$$
$$\wedge \ (y = x - \mathrm{A} \times d) \ \wedge \ (y < 0).$$

---

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

Speed is inertial:
$$\{Speed(s+1)=x\} \leftarrow Speed(s)=x.$$

The *Location* is defined in terms of *Speed* and *Duration* as

$$Location(s+1)=y \leftarrow Location(s)=x \wedge Speed(s)=a \wedge Speed(s+1)=c$$
$$\wedge\, Duration(s)=d \ \wedge \ y=x+((a+c)/2)\times d.$$

2.3 Theorem on Completion

We review the theorem on completion from [4]. The *completion* turns "tight" ASPMT instances into equivalent SMT instances, so that SMT solvers can be used for computing this fragment of ASPMT.

We say that a formula F is in *Clark normal form* (relative to the list $\mathbf{c}$ of intensional constants) if it is a conjunction of sentences of the form

$$\forall \mathbf{x}(G \to p(\mathbf{x})) \tag{1}$$

and

$$\forall \mathbf{x}y(G \to f(\mathbf{x})=y) \tag{2}$$

one for each intensional predicate p and each intensional function f, where $\mathbf{x}$ is a list of distinct object variables, y is a variable, and G is a formula that has no free variables other than those in $\mathbf{x}$ and y, and sentences of the form

$$\leftarrow G. \tag{3}$$

The *completion* of a formula F in Clark normal form (relative to $\mathbf{c}$) is obtained from F by replacing each conjunctive term (1) with

$$\forall \mathbf{x}(p(\mathbf{x}) \leftrightarrow G), \tag{4}$$

each conjunctive term (2) with

$$\forall \mathbf{x}y(f(\mathbf{x})=y \leftrightarrow G), \tag{5}$$

and each conjunctive term (3) with $\neg G$.

An occurrence of a symbol or a subformula in a formula F is called *strictly positive* in F if that occurrence is not in the antecedent of any implication in F.

The *dependency graph* of a formula F relative to $\mathbf{c}$, denoted by $\mathrm{DG}_{\mathbf{c}}[F]$, is the directed graph that

- has all members of $\mathbf{c}$ as its vertices, and
- has an edge from c to d if, for some strictly positive occurrence of $G \to H$ in F, c has a strictly positive occurrence in H, and d has a strictly positive occurrence in G.

We say that F is *tight* on $\mathbf{c}$ if the dependency graph of F relative to $\mathbf{c}$ is acyclic.

Theorem 1 *([4, Theorem 2]) For any formula F in Clark normal form that is tight on* **c**, *an interpretation I that satisfies $\exists xy(x \neq y)$ is a model of* $\mathrm{SM}[F; \mathbf{c}]$ *iff I is a model of the completion of F relative to* **c**.

For example, the car example formalization contains the following implications for the function $Speed(1)$:

$Speed(1) = y \;\leftarrow\; Accel(0) = \text{TRUE} \;\wedge\; Speed(0) = x \;\wedge\; Duration(0) = d \;\wedge\; (y = x + \mathrm{A} \times d)$
$Speed(1) = y \;\leftarrow\; Decel(0) = \text{TRUE} \;\wedge\; Speed(0) = x \;\wedge\; Duration(0) = d \;\wedge\; (y = x - \mathrm{A} \times d)$
$Speed(1) = y \;\leftarrow\; Speed(0) = y \;\wedge\; \neg\neg(Speed(1) = y)$

($\{c = v\} \leftarrow G$ is strongly equivalent to $c = v \leftarrow G \wedge \neg\neg(c = v)$) and the completion contains the following equivalence.

$$
\begin{aligned}
Speed(1) = y \;\leftrightarrow\; & \\
\exists xd(\;\; & (Accel(0) = \text{TRUE} \;\wedge\; Speed(0) = x \;\wedge\; Duration(0) = d \;\wedge\; (y = x + \mathrm{A} \times d)) \\
\vee\; & (Decel(0) = \text{TRUE} \;\wedge\; Speed(0) = x \;\wedge\; Duration(0) = d \;\wedge\; (y = x - \mathrm{A} \times d)) \\
\vee\; & Speed(0) = y \;)
\end{aligned}
\tag{6}
$$

3   Variable Elimination

Some SMT solvers do not support variables at all (e.g. iSAT) while others suffer in performance when handling variables (e.g. Z3). While we can partially ground the input theories, some variables have large (or infinite) domains and should not (or cannot) be grounded. Thus, we consider two types of variables: *ASP variables*—variables which are grounded by ASP grounders—and *SMT variables*—variables which should not be grounded. After eliminating ASP variables by grounding, we consider the problem of equivalently rewriting the completion of the partially ground ASPMT theory so that the result contains no variables.

To ensure that variable elimination can be performed, we impose some syntactic restrictions on ASPMT instances. We first impose that no SMT variable appears in the argument of an uninterpreted function.

We assume ASPMT2SMT programs comprised of rules of the form $H \leftarrow B$ where

– H is $\bot$ or an atom of the form $f(\mathbf{t}) = v$, where $f(\mathbf{t})$ is a term and v is a variable;
– B is a conjunction of atomic formulas possibly preceded with $\neg$.

We define the *variable dependency graph* of a conjunction of possibly negated atomic formulas $C_1 \wedge \cdots \wedge C_n$ as follows. The vertices are the variables occurring in $C_1 \wedge \cdots \wedge C_n$. There is a directed edge from v to u if there is a C_i that is $v = t$ or $t = v$ for some term t such that u appears in t. We say a variable v *depends on* a variable u if there is a directed path from v to u in the variable dependency graph. We say a rule $H \leftarrow B$ is *variable isolated* if every variable v in it occurs in an equality $t = v$ or $v = t$ that is not negated in B and the variable dependency graph of B is acyclic.

Example 1. The rule $f = x \leftarrow g = 2 \times x$ is not variable isolated because variable x does not occur in an equality $x = t$ or $t = x$ in the body. Instead, we write this as $f = x \leftarrow (g = y) \wedge (y = 2 \times x)$, which is variable isolated.

The rule $f = x \leftarrow (2 \times x = y) \wedge (2 \times y = x)$ is not variable isolated; although variable y occurs in an equality of the form $t = y$, the dependency graph is not acyclic.

The variable elimination is performed modularly so the process needs only to be described for a single equivalence. If an ASPMT program contains no variables in arguments of uninterpreted functions, any equivalence in the completion of the ASPMT program will be of the form

$$\forall v(f = v \leftrightarrow \exists \mathbf{x}(B_1(v, \mathbf{x}) \vee \cdots \vee B_k(v, \mathbf{x})))$$

where each B_i is a conjunction of possibly negated literals and has $v = t$ as a non-negated subformula, and the variable dependency graph of B is acyclic. In the following, the notation F_t^v denotes the formula obtained from F by replacing every occurrence of the variable v with the term t. We define the process of eliminating variables from such an equivalence E as follows.

1. Given an equivalence $E = \forall v(f = v \leftrightarrow \exists \mathbf{x}(B_1(v, \mathbf{x}) \vee \cdots \vee B_k(v, \mathbf{x})))$,
   $F := \forall v(f = v \rightarrow \exists \mathbf{x}(B_1(v, \mathbf{x}) \vee \cdots \vee B_k(v, \mathbf{x})))$;
   $G := \forall v(\exists \mathbf{x}(B_1(v, \mathbf{x}) \vee \cdots \vee B_k(v, \mathbf{x})) \rightarrow f = v)$.
2. Eliminate variables from F as follows:
   (a) $F := \exists \mathbf{x}(B_1(v, \mathbf{x})_f^v \vee \cdots \vee B_k(v, \mathbf{x})_f^v)$ and then equivalently,
       $F := \exists \mathbf{x}(B_1(v, \mathbf{x})_f^v) \vee \cdots \vee \exists \mathbf{x}(B_k(v, \mathbf{x})_f^v)$.
   (b) $F_i := \exists \mathbf{x}(B_i(v, \mathbf{x})_f^v)$.
   (c) Eliminate variables from F_i as follows:
       i. $D_i := B_i(v, \mathbf{x})_f^v$.
       ii. While there is a variable x still in D_i, select a conjunctive term $x = t$ or $t = x$ (such that no variable in t depends on x) in D_i, then $D_i := (D_i)_t^x$.
       iii. $F_i = D_i$ (drop the existential quantifier since there are no variables in D_i).
   (d) $F := F_1 \vee \cdots \vee F_k$.
3. Eliminate variables from G as follows:
   (a) $G := \forall v\mathbf{x}((B_1(v, \mathbf{x}) \vee \cdots \vee B_k(v, \mathbf{x})) \rightarrow f = v)$ and then equivalently,
       $G := \forall v\mathbf{x}(B_1(v, \mathbf{x}) \rightarrow f = v) \wedge \cdots \wedge \forall v\mathbf{x}(B_k(v, \mathbf{x}) \rightarrow f = v)$.
   (b) $G_i := \forall v\mathbf{x}(B_i(v, \mathbf{x}) \rightarrow f = v)$.
   (c) Eliminate variables from G_i as follows:
       i. $D_i := B_i(v, \mathbf{x}) \rightarrow f = v$.
       ii. While there is a variable x still in D_i, select a conjunctive term $x = t$ or $t = x$ (such that no variable in t depends on x) from the body of D_i, then $D_i := (D_i)_t^x$.
       iii. $G_i = D_i$ (drop the universal quantifier since there are no variables in D_i).
   (d) $G := G_1 \vee \cdots \vee G_k$.
4. $E := F \wedge G$.

The following proposition asserts the correctness of this method. Note that the absence of variables in arguments of uninterpreted functions can be achieved by grounding ASP variables and enforcing that no SMT variables occur in uninterpreted functions.

Proposition 1 *For any completion of a variable isolated ASPMT program with no variables in arguments of uninterpreted functions, applying variable elimination method repeatedly results in a classically equivalent formula that contains no variables.*

For example, given the equivalence (6), Step 2a) turns the implication from left to right into the formula

$$\exists xd(\quad (Accel(0)\!=\!\text{TRUE} \;\wedge\; Speed(0)\!=\!x \;\wedge\; Duration(0)\!=\!d \;\wedge\; (Speed(1)\!=\!x + \text{A} \times d))$$
$$\vee (Decel(0)\!=\!\text{TRUE} \;\wedge\; Speed(0)\!=\!x \;\wedge\; Duration(0)\!=\!d \wedge (Speed(1)\!=\!x - \text{A} \times d))$$
$$\vee (Speed(0) = Speed(1)))$$

And then step 2d) produces

$$(Accel(0)\!=\!\text{TRUE} \wedge Speed(1)\!=\!Speed(0) + \text{A} \times Duration(0)) \vee$$
$$(Decel(0)\!=\!\text{TRUE} \wedge Speed(1) = Speed(0) - \text{A} \times Duration(0)) \vee$$
$$(Speed(0) = Speed(1)).$$

4   ASPMT2SMT **System**

4.1   **Syntax of Input Language**

In addition to the syntactic restriction on ASPMT rules imposed in the previous section, the current version of system ASPMT2SMT assumes that the input program is **f**-*plain* [5], as well as "av-separated," which intuitively means that no variable occurring in an argument of an uninterpreted function is related to the value variable of another uninterpreted functions via equality.[2] For example, for the rule $f(x) = 1 \leftarrow g = y \wedge y = x$, variable x is an argument of f and is also related to the value variable y of g via equality $y = x$. The reason for this restriction is because the system sets the equalities $g = y$ and $y = x$ aside (so that GRINGO does not ground them), and ground the rule and then replace the equalities back to yield

$$f(1) = 1 \leftarrow g = y \wedge y = x$$
$$f(2) = 1 \leftarrow g = y \wedge y = x$$
$$\dots$$

rather than the intended

$$f(1) = 1 \leftarrow g = y \wedge y = 1$$
$$f(2) = 1 \leftarrow g = y \wedge y = 2$$
$$\dots$$

It should also be noted that the only background theories considered in this version of the implementation are arithmetic over reals and integers.

System ASPMT2SMT uses a syntax similar to system CPLUS2ASP [8] for declarations and a syntax similar to system F2LP [9] for rules.

There are declarations of four kinds, `sorts`, `objects`, `constants`, and `variables`. The sort declarations specify user data types (note: these cannot be used for value sorts). The object declarations specify the elements of the user-declared data types. The constant declarations specify all of the (possibly boolean) function constants that appear in the theory. The variables declarations specify the user-declared data types associated with each variable. Declarations for the car example are shown below.

---

[2] See the system homepage for the precise description of this condition.

```
:- sorts
   step; astep.

:- objects
   0..st            :: step;              0..st-1           :: astep.

:- constants
   time(step)       :: real[0..t];        accel(astep)     :: boolean;
   duration(astep)  :: real[0..t];        decel(astep)     :: boolean;
   speed(step)      :: real[0..ms];       location(step)   :: real[0..l].

:- variables
   S                :: astep;             B                 :: boolean.
```

Only propositional connectives are supported in this version of ASPMT2SMT and these are represented in the system as follows:

$\wedge$	$\vee$	$\neg$	$\rightarrow$	$\leftarrow$
&	|	not	->	<-

Comparison and arithmetic operators are represented as usual:

$<$	$\leq$	$\geq$	$>$	$=$	$\neq$	add	subtract	multiply	divide
<	<=	>=	>	=	!=	+	−	*	/

$a \mathrel{!}= b$ is understood as $\neg(a = b)$. To abbreviate the formula $A \vee \neg A$, which is useful for expressing defaults and inertia, we write $\{A\}$. The rest of the car example is shown below.

```
% Actions and durations are exogenous
{accel(S)=B}.
{decel(S)=B}.
{duration(S)=X}.

% no concurrent actions
<- accel(S)=true & decel(S)=true.

% effects of accel and decel
speed(S+1)=Y <- accel(S)=true & speed(S)=X & duration(S)=D & Y = X+ar*D.
speed(S+1)=Y <- decel(S)=true & speed(S)=X & duration(S)=D & Y = X-ar*D.

% preconditions of accel and decel
<- accel(S)=true & speed(S)=X & duration(S)=D & Y = X+ar*D & Y > ms.
<- decel(S)=true & speed(S)=X & duration(S)=D & Y = X-ar*D & Y < 0.

% inertia of speed
{speed(S+1)=X} <- speed(S)=X.

location(S+1)=Y <- location(S)=X & speed(S)=A &
     speed(S+1)=C & duration(S)=D & Y = X+(A+C)/2*D.
```

```
time(S+1)=Y <- time(S)=X & duration(S)=D & Y=X+D.

% problem instance
time(0)=0.     speed(0)=0.     location(0)=0.
<- location(st) = Z & Z != l.
<- speed(st) = Z & Z != 0.
<- time(st) = Z & Z != t.
```

This description can be run by the command

```
$aspmt2smt car -c st=3 -c t=4 -c ms=4 -c ar=3 -c l=10
```

which yields the output

```
  accel(0) = true  accel(1) = false  accel(2) = false
  decel(0) = false  decel(1) = false  decel(2) = true
  duration(0) = 1.1835034190  duration(1) = 1.6329931618
  duration(2) = 1.1835034190  location(0) = 0.0
  location(1) = 2.1010205144  location(2) = 7.8989794855
  location(3) = 10.0  speed(0) = 0.0
  speed(1) = 3.5505102572  speed(2) = 3.5505102572
  speed(3) = 0.0  time(0) = 0.0  time(1) = 1.1835034190
  time(2) = 2.8164965809  time(3) = 4.0
z3 time in milliseconds: 30
Total time in milliseconds: 71
```
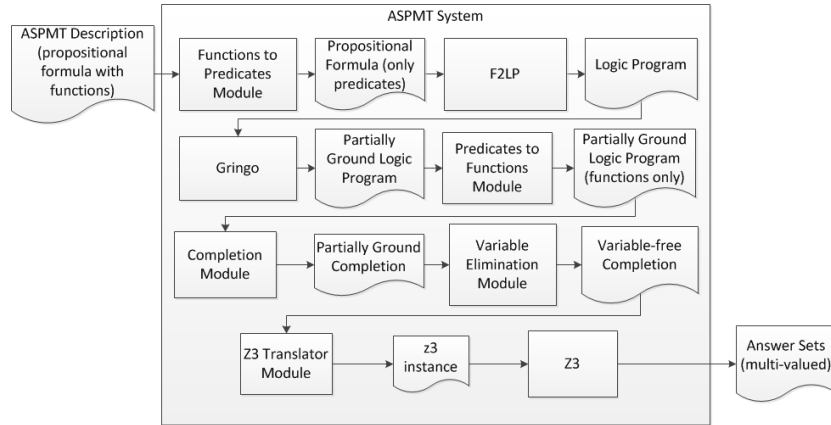
4.2 Architecture

Fig. 1. ASPMT2SMT System Architecture

The architecture of ASPMT2SMT system is shown in Figure 1. The system first converts the ASPMT description to a propositional formula containing only predicates. In addition, this step substitutes auxiliary constants for SMT variables and necessary pre-processing for F2LP (v1.3) and GRINGO (v3.0.4) to enable partial grounding of ASP

variables only. F2LP transforms the propositional formula into a logic program and then GRINGO performs partial grounding on only the ASP variables. The ASPMT2SMT system then converts the predicates back to functions and replaces the auxiliary constants with the original expressions. Then the system computes the completion of this partially ground logic program and performs variable elimination on that completion. Finally, the system converts this variable-free description into the language of Z3 and then relies on Z3 to produce models which correspond to stable models of the original ASPMT description.

For instance, consider the result of variable elimination on the portion of the completion related to $speed(1)$ of the running car example:

$$(Accel(0) = \text{TRUE} \wedge Speed(1) = Speed(0) + A \times Duration(0)) \vee$$
$$(Decel(0) = \text{TRUE} \wedge Speed(1) = Speed(0) - A \times Duration(0)) \vee$$
$$(Speed(0) = Speed(1)).$$

In the language of Z3, this is

```
(assert (or (or
   (and (= accel_0_ true) (= speed_1_ (+ speed_0_ (* duration_0_ a))))
   (and (= decel_0_ true) (= speed_1_ (- speed_0_ (* duration_0_ a)))))
   (= speed_1_ speed_0_) ))
```

5   Experiments

The following experiments demonstrate the capability of the ASPMT2SMT system to perform nonmonotonic reasoning about continuous changes. In addition, this shows a significant performance increase compared to ASP solvers for domains in which only SMT variables have large domains. However, when ASP variables have large domains, similar scalability issues arise as comparable grounding still occurs.

All experiments were performed on an Intel Core 2 Duo 3.00 GHZ CPU with 4 GB RAM running Ubuntu 13.10. The domain descriptions of these examples can be found from the system homepage.

5.1   Leaking Bucket

c	CLINGO v3.0.5 Run Time (Grounding + Solving)	ASPMT2SMT v0.9 Run Time (Preprocessing + solving)
10	0s (0s+0s)	.037s (.027s + .01s)
50	.02s (02s + 0s)	.089s (.079s + .01s)
100	.12s (.12s + 0s)	.180s (.170s + .01s)
500	8.69s (8.68s + .01s)	1.731s (1.661s + .07s)
1000	60.32s (60.29s+ .03s)	35.326s (35.206s + .12s)

Consider a leaking bucket with maximum capacity c that loses one unit of water every time step by default. The bucket can be refilled to its maximum capacity by the action $fill$. The initial capacity is 5 and the desired capacity is 10.

We see that in this experiment, ASPMT2SMT does not yield significantly better results than CLINGO. The reason for this is that the scaling

of this domain takes place in the number of time steps. Thus, since ASPMT2SMT uses GRINGO to generate fluents for each of these time steps, the ground descriptions given to CLINGO and Z3 are of similar size. Consequently, we see that the majority of the time taken for ASPMT2SMT is in preprocessing.

5.2   Car Example

Recall the car example in Section 2.2. The first half of the experiments are done with the values L = $10k$, A = $3k$, MS = $4k$, T = $4k$, which yields solutions with irrational values and so cannot be solved by system CLINGO. The second half of the experiments are done with the values L = $4k$, A = k, MS = $4k$, T = $4k$, which yields solutions with integral values and so can be solved by system CLINGO. In this example, only the SMT variables have increasing domains but the ASP variable domain remains the same.

k	CLINGO v3.0.5 Run Time (Grounding + Solving)	ASPMT2SMT v0.9 Run Time (Preprocessing + solving)
1	n/a	.084s (.054s + .03s)
5	n/a	.085s (.055s + .03s)
10	n/a	.085s (.055s + .03s)
50	n/a	.087s (.047s + .04s)
100	n/a	.088s (.048s + .04s)
1	.61s (.6s + .01s)	.060s (.050s + .01s)
2	48.81 (48.73s + .08s)	.07s (.050s + .02s)
3	> 30 minutes	.072s (.052s + .02s)
5	> 30 minutes	.068s (.048s + .02s)
10	> 30 minutes	.068s (.048s + .02s)
50	> 30 minutes	.068s (.048s + .02s)
100	> 30 minutes	.072s (.052s + .02s)

Consequently, the ASPMT2SMT system scales very well compared to the ASP system which can only complete the two smallest size domains.

We also experimented with CLINGCON. Since CLINGCON does not allow intensional functions, we need to encode the example differently using auxiliary abnormality atoms to represent the notions of inertia and default behaviors. In the first set of experiments, CLINGCON performed better than ASPMT2SMT, but like CLINGO, the current version of CLINGCON cannot handle real numbers, so it is not applicable to the second set of experiments.

5.3   Space Shuttle Example

The following example is from [10], which represents cumulative effects on continuous changes. A spacecraft is not affected by any external forces. It has two jets and the force that can be applied by each jet along each axis is at most $4k$. The initial position of the rocket is (0,0,0) and its initial velocity is (0,1,1). How can

k	CLINGO v3.0.5 Run Time (Grounding + Solving)	ASPMT2SMT v0.9 Run Time (Preprocessing + solving)
1	0s (0s + 0s)	.048s (.038s + .01s)
5	.03s (.02s + .01s)	.047s (.037s + .01s)
10	.14s (.9s + .5s)	.053s (.043s + .01s)
50	7.83s (3.36s + 4.47s)	.050s (.040s + .01s)
100	39.65s (16.14s + 23.51s)	.051s (.041s + .01s)

it get to (0,$3k$,$2k$) within 2 seconds? Assume the mass is 2.

Again in this problem, the scaling lies only in the size of the value of the functions involved in the description. Consequently, we see no scaling issues in either ASPMT2SMT or CLINGCON.

5.4 Bouncing Ball Example

k	CLINGO v3.0.5 Run Time (Grounding + Solving)	ASPMT2SMT v0.9 Run Time (Preprocessing + solving)
1	n/a	.072s (.062s + .01s)
10	n/a	.072s (.062s + .01s)
100	n/a	.071s (.061s + .01s)
1000	n/a	.075s (.065s + .01s)
10000	n/a	.082s (.062s + .02s)

The following example is from [11]. A ball is held above the ground by an agent. The actions available to the agent are *drop* and *catch.* Dropping the ball causes the height of the ball to change continuously with time as defined by Newton's laws of motion. As the ball accelerates towards the ground it gains velocity. If the ball is not caught before it reaches the ground, it hits the ground with speed s and bounces up into the air with speed $r \times s$ where $r = .95$ is the rebound coefficient. The bouncing ball reaches a certain height and falls back towards the ground due to gravity. An agent is holding a ball at height $100k$. We want to have the ball hit the ground and caught at height 50.

Again, CLINGO and CLINGCON are unable to find solutions to this domain since solutions are not integral. Also, we see that ASPMT2SMT suffers no scaling issues here again due to the fact that in this problem the scaling lies only in the size of the value of the functions involved in the description.

6 Conclusion

We presented system ASPMT2SMT, which translates ASPMT instances into SMT instances, and uses SMT solvers to compute ASPMT. Unlike other ASP solvers, this system can compute effective real number computation by leveraging the effective SMT solvers. Future work includes extending the system to handle other background theories, and investigate a larger fragment of ASPMT instances that can be turned into SMT instances.

References

1. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of International Conference on Logic Programming (ICLP). (2009) 235–249

2. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In *Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)* (2009)
3. Janhunen, T., Liu, G., Niemelä, I.: Tight integration of non-ground answer set programming and satisfiability modulo theories. In: Working notes of the 1st Workshop on Grounding and Transformations for Theories with Variables. (2011)
4. Bartholomew, M., Lee, J.: Functional stable model semantics and answer set programming modulo theories. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). (2013)
5. 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
6. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence **175** (2011) 236–263
7. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009) 825–885
8. Babb, J., Lee, J.: Cplus2ASP: Computing action language $\mathcal{C}+$ in answer set programming. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2013) 122–134
9. Lee, J., Palla, R.: System F2LP – computing answer sets of first-order formulas. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2009) 515–521
10. 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
11. Chintabathina, S.: Towards answer set prolog based architectures for intelligent agents. In: AAAI'08. (2008) 1843–1844