2013 International Symposium on Theoretical Aspects of Software Engineering

Integration of Linear Constraints with a Temporal Logic Programming Language

Qian Ma[1] and Zhenhua Duan[1,⋆] and Mengfei Yang[2]
[1] Institute of Computing Theory and Technology P.O. Box 177, Xidian University, Xi'an 710071, P.R.China
[2] Chinese Academy of Space Technology, Beijing 100094, P.R.China
Email: qma@mail.xidian.edu.cn, zhhduan@mail.xidian.edu.cn, yangmf@bice.org.cn

***Abstract*—This paper investigates the integration of linear constraints with MSVL. To this end, we first define linear constraint statements and discuss related issues of the incorporation. Further, for calling SMT solvers to solve the newly introduced constraints, we give a translation algorithm from state programs in MSVL with linear constraints to SMT-LIB2.0 script language and then supply a solving procedure.**

I. Introduction

MSVL [1] is a Modeling, Simulation and Verification language based on Projection Temporal Logic [2]. Due to diverse temporal operators, MSVL can express various constructs, like sequence, branching, while loop, concurrency, and non-determinacy etc. Although as a logic programming language, MSVL also has certain features in the imperative programming style, such as assignment statement, control constructs etc. As a result, MSVL has been successfully applied in specifying and verifying a number of real-life applications, e.g circuits, communicating protocols, many-core parallel computing etc [3].

However, for some applications, like combinatorial optimization, scheduling, hardware/software partitioning problems in embedded systems, MSVL may not deal with them in a suitable way. The reason is that, in such realistic areas, linear constraints are ubiquitous and required, but MSVL lacks constructs to describe these constraints. To enable MSVL to manage such applications, we are motivated to integrate linear constraints into MSVL. The augments of linear constraints into MSVL not only substantially expand the applied range of MSVL, but also make users express their applications in a flexible manner, where some aspects can be specified by constraints while others can be described by imperative constructs. Besides, the incorporation also brings in the extensions of temporal relations with linear constraints. For instance, $\bigcirc(x+y=3)||\bigcirc(x-y=1)$ can be allowed.

The contributions of the paper are as follows: 1. We define linear constraint statements $le_1 \triangle le_2 (\triangle ::= < \mid > \mid = \mid \neq \mid \leq \mid \geq)$, where le_i $(i=1,2)$ is a linear expression and can involve temporal operators $\bigcirc$ and $\circleddash$, and investigate some issues of the integration (Section III). Particularly, to keep compatible with MSVL available, we regard the underlying store as a value-store and address the approach to constraints solving via external SMT solvers [4]. 2. For solving constraints with great generality, we do not designate which SMT solver should be used, but only utilize the common input format SMT-LIB2.0 language that can be flexibly applied in any usable SMT solvers. To do so, we first give a translation algorithm from state programs in MSVL with linear constraints to SMT-LIB2.0 script language, and then supply a solving procedure, which can handle linear optimization problems by SMT solvers (Section IV).

Note that the work presented in [5] introduces linear constraints $=, \leq, \geq$ over rational numbers into MSVL and formalizes the solving procedure by means of the operational semantics. However, some specific issues are not covered in it. For instance, the value of a variable x can be determined by three means: solving constraints, assignment or the framing technique. When x appears in a constraint, which way is used to obtain its value is not discussed. This paper concerns linear constraints $<, >, =, \neq, \leq, \geq$ over integers and addresses the issues of mixing constraints with MSVL in details. Further, we solve constraints by external SMT solvers with a translation algorithm to SMT-LIB2.0 language [6].

The rest of the paper is organized as follows. In Section II, we briefly introduce Projection Temporal Logic and MSVL. Section III defines linear constraint statements and discusses some issues of the incorporation. Further, Section IV provides a translation algorithm to SMT-LIB2.0 script language for invoking SMT solvers to solve constraints. Finally, Section V reviews the related work and Section VI concludes our work.

II. Preliminaries

A. Projection Temporal Logic

Our underlying logic is Projection Temporal Logic (PTL) [2] with infinite models and a new projection construct [7].

1) Syntax: Let $Prop$ be a countable set of atomic propositions and V a countable set of variables. $B = \{true, false\}$ represents the boolean domain while D denotes all data domain needed by us including integers, rational numbers, lists, sets, etc. The terms e and formulas P of PTL are presented by the following grammar:

$$\begin{array}{lll} e & ::= & c \mid v \mid \bigcirc e \mid \circleddash e \mid f(e_1,\dots,e_m) \\ P & ::= & p \mid e_1 = e_2 \mid F(e_1,\dots,e_n) \mid \neg P \mid P_1 \wedge P_2 \mid \\ & & \exists v : P \mid \bigcirc P \mid (P_1,\dots,P_m)\,\mathsf{prj}\,P \mid P^+ \end{array}$$

where $c \in D$ is a constant, $v \in V$, $p \in Prop$, f stands for a function, F indicates a predicate and $P, P_1, \dots, P_m$ are well-formed PTL formulas. A formula (term) is called a *state formula (term)* if it does not contain any temporal operators,

This research is supported by the NSFC Grant Nos. 61133001, 60910004, 61272118, 61272117, 91218301, National Program on Key Basic Research Project (973 Program) Grant No. 2010CB328102. ⋆ Corresponding author.

namely *next* ($\bigcirc$), *previous* ($\ominus$), *projection* (prj), *chop-plus* ($+$); otherwise it is a *temporal formula (term)*. We assume variables are partitioned into static and dynamic variables. A static variable remains the same over an interval whereas a dynamic variable can have different values at different states.

2) Semantics: We define a *state* s over $V \cup Prop$ to be a pair of assignments (I_{var}, I_{prop}) where $s[v] = I_{var}[v]$ for each variable $v \in V$ and $s[p] = I_{prop}[p]$ for each proposition $p \in Prop$. Here $I_{var}[v]$ assigns v a value within a data domain $D' \stackrel{\text{def}}{=} D \cup \{nil\}$ with the appropriate type, in which *nil* means undefined, while $I_{prop}[p]$ sets p a truth value in B. An *interval* $\sigma = \langle s_0, s_1, ...\rangle$ is a non-empty sequence of states, which can be finite or infinite. The length of σ, $|\sigma|$, is the number of states in σ minus one if σ is finite; otherwise it is ω. To have a uniform notation for both finite and infinite intervals, we will use *extended integers* as indices, that is, $N_\omega = N_0 \cup \{\omega\}$ and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$ and for all $i \in N_0, i < \omega$. Moreover, we write $\preceq$ as $\leq -\{(\omega, \omega)\}$.

Let $\sigma = \langle s_0, s_1, \ldots\rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq \ldots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the *projected interval*, $\sigma \downarrow (r_1, \ldots, r_h) \stackrel{\text{def}}{=} \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l}\rangle$, where $t_1, \ldots, t_l$ are attained from $r_1, \ldots, r_h$ by deleting all duplicates. In other words, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. The concatenation($\cdot$) of an interval σ with another interval σ' is represented by $\sigma \cdot \sigma'$ (not sharing any states). To evaluate the existential quantification, an equivalence relation is required. For a variable v, we can write $\sigma' \stackrel{v}{=} \sigma$ if σ' is an interval that is the same as σ except that different values can be assigned to v.

An *interpretation* for a PTL term or formula is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \ldots\rangle$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \preceq j \leq |\sigma|$. We write (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{i,\ldots,j}$ with the current state being s_k. We use I^k_{var} and I^k_{prop} to denote the state interpretation at state s_k. Each *m*-place function symbol *f* has an interpretation $\mathcal{I}[f]$ which is a function mapping *m* elements in D'^m to a single value in D'. Interpretations of predicate symbols $\mathcal{I}[F]$ are similar but map to truth values. We assume that $\mathcal{I}$ interprets operators such as $+, -, *, /$ and $<, >, \leq, \geq, =$ etc standardly. The evaluation of e relative to $\mathcal{I} = (\sigma, i, k, j)$ is defined as $\mathcal{I}[e]$ shown in Figure 1 while the satisfaction relation $\models$ for formulas is given in Figure 2.

$$
\begin{aligned}
&\mathcal{I}[c] = c \in D\\
&\mathcal{I}[v] = s_k[v] \ = \ I^k_{var}[v]\\
&\mathcal{I}[f(e_1, \ldots, e_m)] = \begin{cases} \mathcal{I}[f](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \\ \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } 1 \leq h \leq m & \\ nil \qquad\qquad \text{otherwise} & \end{cases}\\
&\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j\\ nil & \text{otherwise} \end{cases}\\
&\mathcal{I}[\ominus e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k\\ nil & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 1. Interpretation of PTL terms

$$
\begin{aligned}
&\mathcal{I} \models p \ \text{ iff } s_k[p] = I^k_{prop}[p] = \ true\\
&\mathcal{I} \models F(e_1, \ldots, e_n) \ \text{ iff } \mathcal{I}[F](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) = true \text{ and,}\\
&\qquad\qquad\qquad\qquad\quad \mathcal{I}[e_h] \neq nil \text{ for all } 1 \leq h \leq n\\
&\mathcal{I} \models e_1 \ = \ e_2 \ \text{ iff } \mathcal{I}[e_1] \ = \ \mathcal{I}[e_2]\\
&\mathcal{I} \models \neg P \ \text{ iff } \mathcal{I} \not\models P\\
&\mathcal{I} \models P \land Q \ \text{ iff } \mathcal{I} \models P \text{ and } \mathcal{I} \models Q\\
&\mathcal{I} \models \bigcirc P \ \text{ iff } k < j \text{ and } (\sigma, i, k+1, j) \models P\\
&\mathcal{I} \models \exists v \ : \ P \ \text{ iff } (\sigma', i, k, j) \models P \text{ for some } \sigma' \stackrel{v}{=} \sigma\\
&\mathcal{I} \models (P_1, \ldots, P_m) \ \mathsf{prj} \ P \ \text{ iff there exist integers } r_0, \ldots, r_m\\
&\text{and } k = r_0 \leq \ldots \leq r_{m-1} \preceq r_m \leq j \text{ such that}\\
&(\sigma, i, r_0, r_1) \models P_1 \text{ and } (\sigma, r_{l-1}, r_{l-1}, r_l) \models P_l \text{ for all}\\
&1 < l \leq m \text{ and } (\sigma', 0, 0, |\sigma'|) \models P \text{ for } \sigma' \text{ given by :}\\
&(1)\ r_m < j \text{ and } \sigma' \ = \ \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_{m+1}, \ldots, j)}\\
&(2)\ r_m = j \text{ and } \sigma' \ = \ \sigma \downarrow (r_0, \ldots, r_h) \text{ for some } 0 \leq h \leq m\\
&\mathcal{I} \models P^+ \ \text{ iff there are finitely many integers } r_0, \ldots, r_n \text{ and}\\
&k = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \ \preceq r_n = j \ (n \geq 1) \text{ such that}\\
&(\sigma, i, r_0, r_1) \models P \text{ and } (\sigma, r_{l-1}, r_{l-1}, r_l) \models P \text{ for all}\\
&1 < l \leq n.
\end{aligned}
$$

Fig. 2. Interpretation of PTL formulas

A formula *P* is satisfied by an interval σ, signified by $\sigma \models P$ if $(\sigma, 0, 0, |\sigma|) \models P$. A formula *P* is called *satisfiable* if $\sigma \models P$ for some σ. Furthermore, *P* is said to be *valid*, denoted by $\models P$, if $\sigma \models P$ for all intervals σ.

3) Derived formulas: Some derived formulas from elementary PTL formulas are formalized below for the sake of convenience. The abbreviations $\mathsf{true}, \mathsf{false}, \lor, \rightarrow$ and $\leftrightarrow$ are defined as usual while others are explained in [1], [2].

$$
\begin{aligned}
&\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc \mathsf{true} \quad && \mathsf{len}(n) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0\\ \bigcirc \mathsf{len}(n-1) & \text{if } n > 1 \end{cases}\\
&P^* \stackrel{\text{def}}{=} P^+ \lor \varepsilon && P \ ; Q \stackrel{\text{def}}{=} (P, Q)\, \mathsf{prj} \ \varepsilon\\
&\mathsf{more} \stackrel{\text{def}}{=} \neg\ \varepsilon && \Box P \stackrel{\text{def}}{=} \neg\ (\mathsf{true}, \neg P)\, \mathsf{prj} \ \varepsilon\\
&\mathsf{skip} \stackrel{\text{def}}{=} \mathsf{len}(1) && \mathsf{halt}(P) \stackrel{\text{def}}{=} \Box(\varepsilon \leftrightarrow P)\\
& && x := e \stackrel{\text{def}}{=} \mathsf{skip} \land \bigcirc x = e\\
&\multicolumn{2}{l}{\texttt{if } b \texttt{ then } P \texttt{ else } Q \stackrel{\text{def}}{=} (b \rightarrow P) \land (\neg b \rightarrow Q)}\\
&\multicolumn{2}{l}{\texttt{while } b \texttt{ do } P \stackrel{\text{def}}{=} (b \land P)^* \land \Box(\varepsilon \rightarrow \neg b)}\\
&\multicolumn{2}{l}{P \parallel Q \stackrel{\text{def}}{=} (P \land (Q \ ; \mathsf{true})) \lor (Q \land (P \ ; \mathsf{true}))}
\end{aligned}
$$

Sometimes, $\models \Box(P \leftrightarrow Q)$ is represented by $P \equiv Q$ (*strong equivalent*), meaning that P and Q have the same truth at all states in every model.

B. MSVL

MSVL [1], [8] is a modeling, simulation and verification language, which provides an executable PTL framework with more succinct description and immediate practical application. In MSVL, expressions can be regarded as PTL terms while statements can be considered as PTL formulas. The arithmetic expression *e* and boolean expression *b* of MSVL are inductively defined as follows:

$$
\begin{aligned}
e \ &::= c \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ op \ e_1\\
&\quad (op \ ::= \ + \mid - \mid * \mid /)\\
b \ &::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b_0 \land b_1 \mid e_0 = e_1 \mid e_0 < e_1
\end{aligned}
$$

where *c* is a constant, *x* a variable. A dynamic variable *x* is said to be *framed* in program *P* if $\mathsf{frame}(x)$ or $\mathsf{lbf}(x)$ is included in *P* while a program *P* is said to be *framed* if *P* contains at least one framed variable.

Termination :	ε
Unification :	$x\ =\ e$
Positive Immediate Assignment :	$x\ \Leftarrow\ e$
Assignment :	$x :=\ e$
Sequential statement :	$P\ ;\ Q$
State Frame :	$\mathsf{lbf}(x)$
Interval Frame :	$\mathsf{frame}(x)$
Conjuction :	$P \wedge Q$
Selection :	$P \vee Q$
Next statement :	$\bigcirc P$
Always statement :	$\Box P$
Conditional statement:	$\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q$
Existential Quantification :	$\exists x\ : P(x)$
While statement :	$\texttt{while}\ b\ \texttt{do}\ P$
Parallel :	$P \parallel Q$
Projection :	$(P_1, \ldots, P_m)\ \mathsf{prj}\ Q$
Synchronized Communication :	$\mathsf{await}(c)$

Fig. 3. MSVL programs

A framed program in MSVL can be formalized in Figure 3. ε is the termination statement, which simply states that the current state is the final state of the interval over which a program is executed. The next statement $\bigcirc P$ means that P holds at the immediate successive state. $\Box P$ implies that P is always true in all states from now on until the termination of an interval. The sequential statement $P\ ;Q$ signifies a computation of P followed by Q, that is, P keeps executing from the current state until some point in the future at which it terminates and Q will start executing from that point. The conditional statement $\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q$ and while statement $\texttt{while}\ b\ \texttt{do}\ P$ can be illustrated as that in the conventional imperative languages. In the unification statement $x = e$, if e is evaluated to a constant in *D* and x has not been specified at the current state or was specified to the same value as e, we say x is unified with e. The assignment statement $x :=\ e$ claims that at the current state e is first evaluated to a constant and then at the next state x equals the constant, which is executed over an interval with the length 1. $P \vee Q$ represents the selection statement asserting that P or Q is executed with non-determinacy. The existential quantification statement $\exists x : P(x)$ intends to hide the variable x within the process P and may allow a process P to take advantage of a local variable.

The conjunction statement $P \wedge Q$ declares that the processes P and Q are executed concurrently sharing all the states during the mutual execution. The parallel construction $P \parallel Q$ shows another concurrent computation manner. The distinguished difference between $P \parallel Q$ and $P \wedge Q$ is that the former permits both P and Q to be autonomous, or rather to specify their own intervals while the latter does not. Projection can be treated as a special parallel computation with greater autonomy, which is executed on two different time scales. $(P_1, \ldots, P_m)\ \mathsf{prj}\ P$ tells us that P is executed in parallel with $P_1, \ldots, P_m$ over an interval obtained by taking the endpoints of the intervals over which the P_i's are executed. In this construct, processes $P_1, \ldots, P_m, P$ are self-governing and each of them has the right to specify the interval over which it is executed. In particular, the sequence of P_i's and P may terminate at different points.

Framing concerns how the value of a variable can be carried from one state to the next. The crux is how to perceive the assignments of values to variables. To identify an occurrence of an assignment to a variable, say *x*, we make use of a flag called the *assignment flag*, denoted by predicate $\mathsf{af}(x)$; it is $true$ whenever an assignment of a value to *x* is encountered, and $false$ otherwise. To define $\mathsf{af}(x)$, a new assignment *positive immediate assignment* $x\ \Leftarrow\ e \overset{\text{def}}{=} x\ =\ e \wedge p_x$ is needed, where p_x is an atomic proposition connected with variable x and cannot be used for other purposes. Then the assignment flag is formalized as $\mathsf{af}(x) \overset{\text{def}}{=} p_x$. As expected, when $x \Leftarrow e$ is encountered, p_x is set to $true$, hence $\mathsf{af}(x)$ is $true$; whereas if no assignment to x takes place, p_x is unspecified. In this case, we will use the minimal model to force it to be $false$. Note that the definition given above is one way to specify $\mathsf{af}(x)$ and there may be some other methods to formulate it. Armed with $\mathsf{af}(x)$ we can define state frame $\mathsf{lbf}(x) \overset{\text{def}}{=} \neg\mathsf{af}(x) \rightarrow \exists\ a : (\circleddash x = a \wedge x = a)$ and interval frame $\mathsf{frame}(x) \overset{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc\ \mathsf{lbf}(x))$, where a is a static variable. Intuitively, $\mathsf{lbf}(x)$ means that, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state while $\mathsf{frame}(x)$ implies that a variable is framed over an interval if it is framed at each state over the interval.

The framing operator enables us to formalize the synchronizing construct $\mathsf{await}(c) \overset{\text{def}}{=} \mathsf{halt}(c) \wedge \mathsf{frame}(V_c)$, where *c* is a condition, i.e. a boolean expression and V_c represents all dynamic variables contained in *c*. The await statement is employed to synchronize communication between parallel processes with shared variables. It does not change any variables, but waits until the condition *c* becomes true, at which point it terminates.

III. Integration of linear constraints with MSVL

In this section, we incorporate linear constraints into MSVL. Here linear constraints mean linear arithmetic relations $=, \neq, \leq, \geq, <, >$, which are widely used in practice. As a matter of fact, MSVL has certain characteristics in common with imperative programming whereas constraints posses some declarative flavor, which makes some issues be considered when introducing linear constraints into MSVL.

A. Some issues

Underlying store In MSVL, the underlying store (i.e a state) complies with the imperative paradigm and is a value-store based on the Von Neumann memory model, where each variable is assigned a uniquely determined value and changed merely via explicit assignment. On the contrary, constraint usually accompanies with a constraint-store, in which each variable possibly holds a set of values, and adding new constraints or removing existing constraints can modify the value of a variable. To smoothly integrate linear constraints with MSVL, we make a compromise between them. For keeping compatible with MSVL available, the essential interpretation of a state is not altered and still defined as the value-store. When constraints are encountered, we solve them by some means or other and only obtain a single solution at one state.

Constraint statements Let Z be the set of integers. In MSVL with linear constraints, the numeric domain is a subset of Z and assumed to be finite. Further, the data type can be integer (int), boolean, one-dimensional or two-dimensional array, where each element in an array has the same type and is an integer. Motivated to place constraints on variables, firstly linear expressions are defined as follows:

$$\begin{aligned} le \ ::= \ & c \mid x \mid \mathsf{max} \mid \mathsf{min} \mid \bigcirc^n (le) \mid \circleddash x \mid le_1 + le_2 \mid \\ & le_1 - le_2 \mid -(le) \mid c * le \mid le/c \mid le \ \mathsf{mod}\ c \mid \\ & x[n] \mid x[n][m] \end{aligned}$$

where c is a constant, i.e. an integer, n, m are non-negative constants, x stands for a variable (either dynamic or static), $+, -, *, /, \mathsf{mod}$ represent addition, binary substraction or unary negation, multiplication, division and modular arithmetic respectively as usual. $x[n]$ states the n^{th} element in the one-dimensional array x is accessed while $x[n][m]$ indicates the element in the n^{th} row and m^{th} column in the two-dimensional array x is fetched. max and min are two special variables, which are only concerned with optimization problems as well as the maximum and minimum objective statement respectively. In a linear optimization problem, the maximum (resp. minimum) objective statement or function, $\mathsf{max}(\mathsf{min}) \Leftarrow le$, contains max (resp. min) in the left and intends to maximize (resp. minimize) the objective expression le under other constraints.

A linear expression is called a *state linear expression* if it does not contain any temporal operators, i.e., $\bigcirc$ or $\circleddash$; otherwise it is a *temporal linear expression*. Since the current version of MSVL only involves several data types and we merely care for linear constraints in this paper, the definition for linear expressions is sufficient for us. Based on linear expressions, linear constraint statements are formalized as:

Linear Constraints: $le_1 \triangle le_2$ $(\triangle ::= < \mid > \mid = \mid \neq \mid \leq \mid \geq)$

where le_1 and le_2 stands for linear expressions. The meaning of $le_1 \triangle le_2$ is intuitive. For example, $le_1 < le_2$ claims that the value of le_1 is less than the value of le_2. Others can be illustrated in a similar manner. Actually, in MSVL with linear constraints, $le_1 \triangle le_2$ plays two roles: comparison and constraint. When it appears in a boolean expression associated with conditional or iterative statements, $le_1 \triangle le_2$ acts as comparison whereas when it occurs as an independent statement, $le_1 \triangle le_2$ works as a constraint and needs to be solved. Especially, although the unification $x = c$ is a constraint, after solving it the solution of x is still c. Thus, it can be regarded as a special assignment.

In MSVL with linear constraints, these linear arithmetic relations can be transformed each other conventionally. For instance, $le_1 \leq le_2 \equiv le_2 \geq le_1 \equiv -le_1 \geq -le_2 \equiv le_1 < le_2 + 1 \equiv le_2 + 1 > le_1 \equiv (le_1 < le_2) \vee (le_1 = le_2)$. Some compound constructs are presented in the following, which enables us to write concise programs. $x \in [c_1, c_2]$ states that the value of x is greater than or equal to c_1 and less than or equal to c_2. When $c_1 > c_2$, the formula is false. The meaning for $c_1 \geq x \geq c_2$ is the same as $x \in [c_2, c_1]$ and formalized here for flexible use. Other abbreviations like $c_1 \leq x \leq c_2, x \in (c_1, c_2]$ can also be defined in an analogous fashion. $\mathsf{allequal}(le_1, \ldots, le_n)$ is a multi-equality and implies all these linear expressions are equal each other whereas $\mathsf{alldiffer}(le_1, \ldots, le_n)$ indicates any le_i and le_j are pairwise different as long as $i \neq j$.

$$\begin{aligned} & x \in [c_1, c_2] \overset{\text{def}}{=} x \geq c_1 \wedge x \leq c_2 \\ & c_1 \geq x \geq c_2 \overset{\text{def}}{=} c_1 \geq x \wedge x \geq c_2 \\ & \mathsf{allequal}(le_1, \ldots, le_n) \overset{\text{def}}{=} le_1 = le_2 \wedge \ldots \wedge le_1 = le_n \\ & \mathsf{alldiffer}(le_1, \ldots, le_n) \overset{\text{def}}{=} le_1 \neq le_2 \wedge le_1 \neq le_3 \wedge \ldots \wedge le_1 \neq \\ & \qquad le_n \wedge le_2 \neq le_3 \wedge \ldots \wedge le_2 \neq le_n \wedge \ldots \wedge le_{n-1} \neq le_n \end{aligned}$$

where $le_1, \ldots, le_n$ are state linear expressions

How to determine values of variables With the presence of linear constraints in MSVL, the value of a variable in a state can be acquired by the framing technique, an assignment statement or constraints solving. Generally, in a constraint which does not contain $\bigcirc$, variables with $\circleddash$ are substituted with their previous values via the framing method while others are determined via solving constraints. For instance, in $\mathsf{lbf}(z) \wedge x + y = \circleddash z$, assume z is equal to 3 at the previous state. Firstly $\circleddash z$ is evaluated into 3, and then the constraint $x + y = 3$ is solved to get values of x and y. Further, when a constraint involves $\bigcirc$, variables without next operators are replaced by their current values, which can be attained through any above approaches. E.g. $\bigcirc x < y$ is considered as $\bigcirc x < c$ in programs $y = c \wedge \bigcirc x < y$, $\mathsf{lbf}(y) \wedge \bigcirc x < y$ with $\circleddash y = c$, and $y + z = c + 1 \wedge y - z = c - 1 \wedge \bigcirc x < y$, where the values of y are respectively obtained by assignment, framing and solving constraints. Normally, as boolean expressions and the right side of $:=$ within an assignment statement are not concerned with constraint solving, variables in them are treated as the substitution of their current values.

Constraint solving After the introduction of linear constraints into MSVL, it is indispensable to solve these constraints and produce concrete values for reaching the value-store. Hence, a constraint solver is needed to be responsible for this function. Typically, there are two methods to obtain such a constraint solver: one is to implement it from scratch and the other is to invoke the existing constraint solvers. With the first approach, we can clearly understand how to solve constraints at length and do not translate our language to the specific language for the external solver, which might be intricate sometimes. However, for different types of constraints, like integer, rational numbers, array etc, we have to enforce different algorithms, which takes much time, suffers from hard extensions and makes us be trapped in details of solving. On the contrary, the second technique shortens the duration of this process and facilitates the future development of MSVL, particularly the diverse data types, but bears a burden of mapping between two different languages.

In order to efficiently utilize MSVL with linear constraints, we plan to carry out these two means and evaluate which one is better suited. In fact, [5] took the first approach for solving linear constraints $=, \leq$ and $\geq$ over rational numbers. Therefore, in this paper, we explore the second method and employ outside SMT solvers [4], [9], [6] as the background constraint solver due to their theories maturity and practice applications.

B. Semi-normal form and normal form

To deal with MSVL programs with linear constraints in a unified way, we define semi-normal form (SNF) and normal form (NF) respectively. The reason for formalizing the two kinds of forms rests in that SNF makes us get rid of the constraint solving and investigate programs in a more abstract level whereas the reduction of MSVL programs with linear constraints is actually based upon NF.

Definition 1 (Semi-Normal Form, SNF): Let Q be an MSVL program with linear constraints. The semi-normal form of Q is defined as

$$Q \overset{\text{def}}{=} \bigvee_{i=1}^{k}(Q_{ei} \wedge \varepsilon) \vee \bigvee_{j=1}^{h}(Q_{cj} \wedge Q_{mj} \wedge \bigcirc Q_{fj})$$

where $k + h \geq 1$. Q_{fj} is a general program, that is, one in which variables may refer to the previous states but not beyond the first state of the current interval over which the program is executed; each Q_{ei} and Q_{cj} is either true or a state formula of the form : $P_1 \wedge \ldots \wedge P_m$ $(m \geq 1)$ such that each P_l $(1 \leq l \leq m)$ is either $le_1 \triangle\ le_2$ with le_1 and le_2 state linear expressions, or $\dot{p}_{x_l}$ denoting p_{x_l} or $\neg p_{x_l}$; each Q_{mj} is either true or a formula of the form : $P_1 \wedge \ldots \wedge P_n$ $(n \geq 1)$ such that each P_l $(1 \leq l \leq n)$ is $\bigcirc le \triangle\ ls$ with le a linear expression and ls a state linear expression.

In some cases, we simply write $Q_e \wedge \varepsilon$ instead of $\bigvee_{i=1}^{k}(Q_{ei} \wedge \varepsilon)$. Ordinarily, if Q terminates at the current state it is reduced to $Q_e \wedge \varepsilon$; otherwise it is reduced to $Q_{cj} \wedge Q_{mj} \wedge \bigcirc Q_{fj}$. The conjuncts $Q_e \wedge \varepsilon$ and $Q_{cj} \wedge Q_{mj} \wedge \bigcirc Q_{fj}$ are named *basic terminal product* and *basic future products* respectively. Moreover, Q_{cj} (or Q_e), Q_{mj} and $\bigcirc Q_{fj}$ are called *present components*, *mixture components* and *future components* respectively. Normally, all the variables in ls within mixture components explicitly appear in present components.

Definition 2 (Normal Form, NF): Let Q be an MSVL program with linear constraints. The normal form of Q is defined as

$$Q \overset{\text{def}}{=} \bigvee_{i=1}^{k}(Q_{ei} \wedge \varepsilon) \vee \bigvee_{j=1}^{h}(Q_{cj} \wedge \bigcirc Q_{fj})$$

where $k+h \geq 1$. Q_{fj} is a general program; each Q_{ei} and Q_{cj} is either true or a state formula of the form : $P_1 \wedge \ldots \wedge P_m$ $(m \geq 1)$ such that each P_l $(1 \leq l \leq m)$ is either $x = c$ with $x \in V, c \in D$, or $\dot{p}_{x_l}$ denoting p_{x_l} or $\neg p_{x_l}$.

The terminologies for SNF are also fit for NF. When a program has been in SNF, we can invoke SMT solvers to solve *present components* and obtain one solution in the form of $x_1 = c_1 \wedge \ldots \wedge x_r = c_r$. After this, the state expression ls in *mixture components* can be replaced by its value. Then mixture components can be transformed into an equivalent statement equipped with $\bigcirc$ at the outermost level, which can be further combined with *future components* in SNF and constructs new future components. In this way, a program in SNF can be reduced into another program in NF. Particularly, when the domain is finite, the solutions of a present component in SNF are finite and the disjunction of these solutions is congruent with the present component, which enables a program in SNF to be rewritten into a logically equivalent form in NF.

Theorem 1: For any satisfiable program P over the finite domain in MSVL with linear constraints, there exists a semi-normal form R and normal form Q such that $P \equiv R \equiv Q$.

From above, we can see that the actual interaction between the integrated declarative constraints and the imperative characteristics in MSVL arises in the solving procedure for present components after translating a program into its SNF. For clarity, we formalize a notation sp called *state programs* as $\mathsf{sp} ::= \mathsf{true} \mid le_1 \triangle le_2 \mid \mathsf{lbf}(x) \mid \mathsf{sp}_1 \wedge \mathsf{sp}_2$, where all the linear expressions are state linear expressions. Actually, as $\mathsf{lbf}(x)$ can be reduced into $x = \odot x$, state programs correspond to the present components Q_{cj} and Q_e in the semi-normal form.

IV. Solving constraints with SMT solvers

In this section, we call the state-of-art SMT solvers [10], [11] as the external constraint solvers, which are required to support linear constraints. For instance, Z3 [9] and CVC3 [12] are such SMT solvers. For great generality, we do not designate which SMT solver ought to be used, but only employ the standard input format SMT-LIB language [6] that can be applied to any usable SMT solvers in a flexible fashion. From above, we know that only the present components in SNF need to be solved. Thus, firstly we provide a translation from present components to SMT-LIB2.0 language and then give a solving procedure Solve to deal with the newly introduced constraints.

A. Satisfiability modulo theory

As an extension of SAT, Satisfiability Modulo Theory (SMT) [4] problem is to determine if a given logic formula expressed in (quantifier-free) first-order logic is satisfiable in one or more theories. Since its birth, SMT has been used in diverse practical areas, like verification, type inference, static program analysis and scheduling etc. One of the distinguished features of SMT is the treatment for several theories in combinations, which avoids encoding everything in one theory. Abundant theories are involved in SMT, such as uninterpreted functions, linear arithmetic, arrays, lists, bit vectors etc. In this paper, we are interested in equality, linear integer arithmetic and array. Therefore, the logic *QF_AUFLIRA* in SMT is enough for us, which involves Quantifiers-Free theories of Array, Uninterpreted Functions and Linear Integer/Real Arithmetic. Currently, numerous SMT solvers [12], [11], [9], [13] have been developed and are available. To evaluate their efficiency uniformly with a common language, SMT-LIB [6] is put forward as the standard input format and benchmark language. Particularly, the command script language in SMT-LIB2.0 describes a script either in a file or at a input prompt, and is able to incrementally assert and retract sets of formulas, which possibly brings in faster running time. Hence, we take advantage of it as the target language, where the syntax is prefix in style, and automatically produce a script as an individual file via mapping.

B. Translation to SMT-LIB2.0

In the following, we assume that readers are familiar with the syntax of SMT-LIB2.0. The translation from present components in MSVL with linear constraints to SMT-LIB 2.0 language concerns the mappings of data types, linear expressions and present components, and the generation of a script file.

Data types Since current version of MSVL with linear constraints merely contains simple data types, all of which have been involved in SMT, the translation for data types is straightforward. Concretely, 'Int' and 'boolean' are respectively mapped into 'Int' and 'Bool'. For an array with integer elements, we separate it into several integer elements and map their data types into 'Int' rather than 'Array', which is because the use of 'Array' in SMT is not very efficient for us.

Linear expressions First of all, we define a function Ide from MSVL symbols to SMT-LIB2.0 identifiers as follows:

$$\mathsf{Ide}: \{+ \mapsto +, - \mapsto -, * \mapsto \star, / \mapsto \mathsf{div}, \mathsf{mod} \mapsto \mathsf{mod}, \\ \leq \mapsto <=, \geq \mapsto >=, < \mapsto <, > \mapsto >, = \mapsto =, \neq \mapsto \mathsf{distinct}\}$$

During the translation, only state linear expressions need to be taken into account, thus a recursive mapping over the syntactic structure of expressions can be formalized as Texp below. Therein, *op* represents $+, -, *, /, \mathsf{mod}$ and we specially treat $+$ for multiple expressions since it is handily dealt in SMT-LIB. *id* denotes constants and variables, such as $5, \mathsf{max}, \mathsf{min}, x$, which are directly translated into identifiers in SMT with same names. Further, the elements in arrays are mapped into variables in SMT with new names.

```
Function $\mathsf{Texp}(le)$
  case
    $le$ is $c$: return $c$;
    $le$ is $x[n]$: return $x\_n$;
    $le$ is $x[n][m]$: return $x\_n\_m$;
    $le$ is $id$: return $\mathsf{id}$;
    $le$ is $-(le_1)$: return (- Texp ($le_1$));
    $le$ is $le_1\ op\ le_2$: return ($\mathsf{Ide}(op)$ $\mathsf{Texp}(le_1)$ $\mathsf{Texp}(le_2)$);
    $le$ is $le_1 + \ldots + le_n$: return (+ $\mathsf{Texp}(le_1) \ldots \mathsf{Texp}(le_n)$);
  end case
```

Present components

```
Function $\mathsf{Tfor}(\mathsf{sp})$
  case
    sp is true: return $true$;
    sp is $le_1 \triangle le_2$: return ($\mathsf{Ide}(\triangle)$ $\mathsf{Texp}(le_1)$ $\mathsf{Texp}(le_2)$);
    sp is $\mathsf{allequal}(le_1, \ldots, le_n)$:
        return (= $\mathsf{Texp}(le_1) \ \ldots \ \mathsf{Texp}(le_n)$);
    sp is $\mathsf{alldiffer}(le_1, \ldots, le_n)$:
        return (distinct $\mathsf{Texp}(le_1) \ \ldots \ \mathsf{Texp}(le_n)$);
    sp is $P_1 \wedge \ldots \wedge P_n$: return (and $\mathsf{Tfor}(P_1) \ldots \mathsf{Tfor}(P_n)$);
    sp is $P_1 \vee \ldots \vee P_n$: return (or $\mathsf{Tfor}(P_1) \ldots \mathsf{Tfor}(P_n)$);
  end case
```

How to translate present components (i.e state programs) into SMT-LIB2.0 format is given in algorithm Tfor, which proceeds by a recursion over the statement structure. Although '$\vee$' are not considered in the definition of sp, its translation is also presented here. This is useful in such cases when one might expect to get another solution after failures of some solutions, which must be excluded through the conjunction of sp with disjunctions of $x_i \neq c_i$. The reason for special mapping alldiffer and allequal is also on account of the convenient management of SMT-LIB.

Script file On the basis of the translation of state programs, a script file is generated in order to run with any usable SMT solvers. Besides the translated assertions, some extra necessary information like setting the underlying logic etc, also should be automatically added into the script for a successful execution. A pseudo-code of producing such a script is shown in $\mathsf{MSVLtoSMTLIB}$, where $\mathsf{Var}(\mathsf{sp})$ stands for the set of all variables appearing in sp. In the SMT-LIB2.0, all the variables used in assertions must be declared before utilizing, so we first identify variables and their types in a formula. After that we write related setting information and declarations as well as the mapped assertions into the file *script.smt2* by the function write, which takes a file name and a string to be recorded in the file as parameters, and can write some content following others having existed.

```
Function $\mathsf{MSVLtoSMTLIB}(\mathsf{sp})$
  $\{x_1, \ldots, x_r\} = \mathsf{Var}(\mathsf{sp})$;
  for $j = 1$ to $r$
    if ($x_j$ is $x[n]$) then $\langle x_j, type_j\rangle := \langle x\_n, \mathsf{Int}\rangle$;
    else if ($x_j$ is $x[n][m]$)    then
               $\langle x_j, type_j\rangle := \langle x\_n\_m, \mathsf{Int}\rangle$;
         else $\langle x_j, type_j\rangle := \langle x_j, \mathsf{Int}\rangle$;
  $Q$ := $\mathsf{Tfor}(\mathsf{sp})$;
  script.smt2:=write(script.smt2,(
          (set-option :print-success true)
          (set-option :produce-models true)
          (set-option :produce-proofs true)
          (set-logic QF_AUFLIRA)
          (declare-fun $x_1$ () $type_1$)
          ...
          (declare-fun $x_r$ () $type_r$)
          (assert ($Q$))
          (check-sat)
          (get-value ($x_1$ ... $x_r$)))
          );
  return script.smt2
```

From the aforesaid recursive procedures, we can easily see that the translation is clear, which guarantees the soundness of the mapping.

C. Constraints solving procedure with SMT solvers

Now we can feed the produced script to any suitable SMT solvers and obtain the solutions of state programs. Further, we also expect to deal with linear optimization problems, which are not concerned and directly supported by SMT-LIB2.0 language and most SMT solvers at present. Hence, this subsection is devoted to coping with this problem and presenting a constraint solving algorithm.

In principle, the procedure for acquiring the optimization value in linear optimization problems can be regarded as a combination of iterative calls of SMT solvers and a binary search over the values that the objective function can take. Since the minimization problem can be reduced into a maximization problem with the negation of the objective expression, we only care for how to get the maximum value, which is given as $\mathsf{Solvemax}$. Thereinto, $value$ denotes the current value of the objective expression and $lower$ and $higher$ signify a lower and upper bound of the optimization value respectively. $candidate$ represents a potential optimization value, which determines the range of optimization value together with $lower$ and $higher$. Further, the function SMT-solver(*script.smt2*) means invoking SMT solvers to run the script file *script.smt2* as well as obtaining a result, which can be 'sat' or 'unsat', and a model, which is in the form of $x_1 = c_1, \ldots, x_r = c_r$ ($\{x_1, \ldots, x_r\} = \mathsf{Var}(P)$). $model.\mathsf{max}$ attains the value of max in a model and $floor(c)$ is employed to round down c. Note that the binary search for solving optimization problems with SMT solvers is somewhat similar to that in [14], but ours supplies a distinct way to modify the bounds and specifies the procedure based on the script language.

```
Function Solvemax($P$)
  script.smt2:=MSVLtoSMTLIB($P$);
  $\langle result, model\rangle$:= SMT-solver(script.smt2);
  if ($result == unsat$)    then return $\langle unsat, \emptyset\rangle$;
  else {
    $value := model.\mathsf{max}$;
    $lower := value, higher := \infty, candidate := value$;
    while  ($higher == \infty$ or $candidate \neq lower$) {
      script.smt2 := write(script.smt2,((push 1)
      (assert (> max  $candidate$)) (check-sat)
      (get-value ($x_1, \ldots, x_n$))));
      $\langle result, model\rangle$:= SMT-solver(script.smt2);
      if ($result == sat$)    then {
        $value := model.\mathsf{max}$;
        if ($higher == \infty$)    then {
          if ($value < 0$)    then $candidate := 0$;
          else  $candidate := 2 * value$;
          $lower := value$; }
        else {$candidate := value + floor(\frac{higher-value+1}{2})$;
              $lower := value$;}}
      else {
        script.smt2 := write(script.smt2,(pop 1));
        $higher := candidate$;
        $candidate := lower + floor(\frac{candidate-lower+1}{2})$;}}
    return $\langle sat, model\rangle$; }
```

Firstly the algorithm attempts to seek out an initial feasible solution. If this fails, P is unsatisfiable and Solvemax(P) returns $\langle unsat, \emptyset\rangle$. Otherwise, an additional assertion $\mathsf{max} > candidate$ is added into the script file to get another feasible value of max, which might be bigger than the last value. If such a solution is found, when the upper bound is undetermined, $candidate$ is doubly increased until a upper bound is obtained, at which point $candidate$ is modified according to the distance between the upper bound and the current feasible value. Or else, there does not exist such a solution, the values of $higher$ and $candidate$ are changed to shorten the possible range where the optimization value lies in. This process is repeated for several times until $candidate = lower$ and $higher \neq \infty$. Then the maximum value of the objective expression equals $candidate$ and the corresponding model is attained. In accordance to these, we can see that in order to iteratively get the optimization value, assertions like $\mathsf{max} > candidate$ need to be added and removed continuously. Owing to the use of the script language, we can perform this by the command 'push' and 'pop' in an incremental fashion, which avoids executing from scratch again and again and improves the efficiency.

The constraint solving procedure for MSVL with linear constraints based on SMT is demonstrated in Solve, which carries out according to the structures of state programs and returns the tuple $\langle result, model\rangle$. $\mathsf{af}(\mathsf{max}) == true$ indicates $\mathsf{max} \Leftarrow le$ appears in P, which further implies the problem we encountered is about linear optimization. In the minimizing case, the objective expression is assumed to be le and P' is acquired by eliminating the objective statement from P as well as Solvemax is called. Therefore, we can deal with all the newly introduced constraints with SMT.

```
Function Solve($P$)
  if ($\mathsf{af}(\mathsf{max}) == true$)
  then $\langle result, model\rangle$:=Solvemax($P$);
  else if ($\mathsf{af}(\mathsf{min}) == true$)    then
          $\langle result, model\rangle$:=Solvemax($P' \wedge \mathsf{max} = -le$);
       else { script.smt2:=MSVLtoSMTLIB($P$);
              $\langle result, model\rangle$:=SMT-solver(script.smt2); }
  return $\langle result, model\rangle$
```

Equipped with linear constraints, MSVL can specify the production scheduling in [5] and the coin problem in [15] as well as solve them by means of SMT solvers. Also, the train packing problem can be well handled in MSVL with linear constraints, which is described as follows: *Suppose we want to transport two kinds of goods, say g_1 and g_2, from place P_1 to P_2. For each piece of goods g_i ($1 \leq i \leq 2$), its weight is w_i and number is N_i. Now we have n trucks of different types to load these goods. The costs, like oil charge, maintenance charge etc, are different for distinct trucks, which can be calculated beforehand. By the descending order of costs, the trucks are arranged as, say $C_1, \ldots, C_n$, with the capacities $c_1, \ldots, c_n$ respectively. In order to make the total cost as minimum as possible during the transport, it is expected to use fewer trucks, that is a truck needs to pack maximum goods not beyond its capacity. Then we require a feasible scheduling to achieve the goal.* The details are omitted due to the limited space and can be referred to *http://ictt.xidian.edu.cn/qm/casestudy.pdf.*

V. Related work

Embedding constraints into a logic programming language is not a new thing. As earlier as 1980's, Constraint Logic Programming (CLP) [16], [17] has been proposed, which uses constraints solving instead of unification in logic programming. A variety of such languages have flourished [18], [19], [20]. Further based on CLP, Temporal CLP (TCLP) [21], [22], [23], [24] is put forward for dealing with time-dependent constraints, which integrates constraints and temporal logic in a unified framework. Although MSVL is a temporal logic programming language, it is different from the TCLP languages aforementioned since they investigate temporal logic programming from distinct views. In general, TCLP uses the Prolog-like syntax and retains the declarative features of logic programming, which treats the execution of programs as the deduction in tractable logics whereas MSVL combines temporal logic and imperative programming, which considers the execution of programs as the construction of Kripke models for the program formulas.

Some contributions to mixing constraints and imperative programming bring about the languages Kaleidoscope [25], Alma-0 [26], Turtle [27] and Kaplan [14] etc. Besides developing a new language, constraints can also be implemented in existing imperative languages by constraint solving toolkits [28]. Compared with this kind of languages, our host language

MSVL is not just an imperative language. Most prominently, it is a temporal logic programming language and brought up as a formalism for specification and verification for concurrent systems. Further, our work is more attractive in handling time-dependent constraints owing to plentiful temporal operators. However, in this paper, constraints in MSVL are restricted to be linear while constraints can be expressed more freely in the aforesaid languages.

VI. Conclusion

Since linear constraints are solved by external solvers step by step, the efficiency of the reduction in MSVL with linear constraints is primarily dominated by the performance of solving declarative constraints. Although we have employed the script language incrementally, we still need to investigate how to improve the efficiency, particulary for solving linear optimization problems by SMT solvers in the future. Further, an interpreter for MSVL with linear constraints calling SMT solvers will be developed. Moreover, we will do more case studies for practicality and combine other constraints like quadratic constraints, with MSVL for further extensions. Besides, we will also compare MSVL with constraints with some semi-formal modeling languages such as SOFL [29], and explore the verification method [30] of MSVL with constraints.

References

[1] Z. Duan, *Temporal Logic and Temporal Logic Programming Language*. Science Press, 2006.

[2] D. Zhenhua, "An extended interval temporal logic and a framing technique for temporal logic programming," Ph.D. dissertation, University of Newcastle Upon Tyne, May.

[3] N. Zhang, Z. Duan, and C. Tian, "A cylinder computation model for many-core parallel computing," *Theor Comput Sci*, vol. DOI:10.1016/j.tcs.2012.02.011, 2012.

[4] L. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, pp. 69–77, 2011.

[5] Q. Ma and Z. Duan, "Linear time-dependent constraints programming with MSVL," *Journal of Combinatorial Optimization,*, vol. DOI:10.1007/s10878-012-9551-2, 2012.

[6] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[7] Z. Duan, M. Koutny, and C. Holt, "Projection in temporal logic programming," in *Proceedings of Logic Programming and Automated Reasoning*, vol. LNAI 822, 1994, pp. 333–344.

[8] X. Yang, Z. Duan, and Q. Ma, "Axiomatic semantics of projection temporal logic programs," *Mathematical Structures in Computer Science*, vol. 20, no. 5, pp. 865–914, 2010.

[9] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of TACAS08*, vol. LNCS 4963, 2008.

[10] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, "6 years of SMT-COMP," *Journal of Automated Reasoning*, pp. 1–35, 2012.

[11] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proceedings of the 16th International Conference on Computer Aided Verification,CAV'06*, vol. LNCS 4144, 2006, pp. 81–94.

[12] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification,CAV'07*, vol. LNCS 4590, 2007, pp. 298–302.

[13] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *Proceedings of TACAS 2013*, vol. 7795 of LNCS, 2013.

[14] A. S. Köksal, V. Kuncak, and P. Suter, "Constraints as control," in *Proceeding of POPL'12*, 2012, pp. 151–164.

[15] K. R. Apt, *Principles of Constraints Programming*. Massachusetts: Cambridge University Press, 2003.

[16] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 111–119.

[17] P. V. Hentenryck, *Constraint Satisfaction in Logic Programming*. Massachusetts: The MIT Press, 1989.

[18] A. Colmerauer, "An introduction to Prolog III," *Communications of the ACM*, vol. 33, pp. 69–90, 1990.

[19] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, "The constraint logic programming language CHIP," in *Proceedings of FGCS'88*, 1988, pp. 693–702.

[20] M. G. Wallace, S. Novello, and J. Schimpf, "ECL^{i}PS^e: A platform for constraint logic programming," *ICL Systems Journal*, vol. 12, no. 1, pp. 159–200, 1997.

[21] T. Hrycej, "Temporal prolog," in *proceedings of ECAI 88*, 1988, pp. 296–301.

[22] C. Brzoska, "Programming in metric temporal logic," *Theoretic Computater Science*, vol. 202, no. 1-2, pp. 55–125, 1998.

[23] T. W. Frhwirth, "Temporal annotated constraint logic programming," *J. Symb. Comput.*, vol. 22, no. 5/6, pp. 555–583, 1996.

[24] A. Analyti and I. Pachoulakis, "Temporally annotated extended logic programs," *Int J Res Rev Artif Intell*, vol. 2, no. 1, pp. 107–112, 2012.

[25] B. N. Freeman-Benson, "Kaleidoscope: mixing objects, constraints, and imperative programming," *ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 77–88, 1990.

[26] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf, "Alma-0: An imperative language that supports declarative programming," *TOPLAS*, vol. 20, pp. 1014–1066, 1998.

[27] M. Grabmüller and P. Hofstedt, "Turtle: a constraint imperative programming language," in *In Proceedings of the 23 SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, 2003.

[28] O. Krzikalla, "Constraint imperative programming with C++," in *Proceedings of the workshop on declarative programming in the context of object-oriented languages*, 2003, pp. 117–130.

[29] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "Sofl: A formal engineering methodology for industrial applications," *IEEE Trans. Software Eng.*, vol. 24, no. 1, pp. 24–45, 1998.

[30] S. Qin, A. Chawdhary, W. Xiong, M. Munro, Z. Qiu, and H. Zhu, "Towards an axiomatic verification system for javascript," in *TASE11*, 2011, pp. 133–141.