An operational model for the SCLP language

Stefano Bistarelli* Elvinia Riccobene†

Abstract

We present an abstract model for the operational semantics of the *SCLP* language (an extension of the Constraint Logic Programming formalism able to handle Semiring-based Constraint Systems) using the *Gurevich's Abstract State Machines* (ASMs). The proposed model can be taken as a basis for a real implementation of the language. This can be reached through correct stepwise refinements of ASMs down to a model which can be mechanically translated into an executable code.

1 Introduction

The *Semiring-based Constraint Logic Programming* (SCLP) language has been defined in [6] as an extension of the Constraint Logic Programming (CLP) formalism to represent and solve Semiring-based Constraint Satisfaction Problems (SCSP)[5, 7]. SCLP shows to be a very expressive and natural formalism to specify many kinds of real-life problems, specially when the knowledge is not completely available nor crisp.

A model-theoretic, fix-point and operational semantics for SCLP is given in [6], but a real implementation of the language does not exist as well as no design studies have been yet developed.

In this paper we develop a simple interpreter for SCLP programs using the *Gurevich's Abstract State Machines* (ASMs) (previously called *evolving algebras* [4]). This formalism has been applied successfully to specify real programming languages and architectures, to validate standard language implementations, to verify real-time protocols, etc. (see annotated ASM bibliography in [2]).

We specify an operational semantic model that directly reflects the intuitive procedural understanding of SCLP programs, but is formulated at the level of abstract search spaces. This combination of procedural and abstract features, provides a tool for mathematical description and analysis of the design decisions for the language, in a machine and proof system independent way. Moreover, it lays the ground for provably correct stepwise refinement, through a hierarchy of specifications at lower level, down to implementations. Therefore, the ASM model we propose can be viewed as the first step versus a real implementation of the language: according to the result presented in [1], that can be reached through a sequence of (correctly) refined ASMs down to a model which can be mechanically translated into an executable code.

The model also suggests a possible strategy to optimize the parallel computation of all possible solutions, formalizing useful techniques which allow to avoid infinite and/or non optimal computations.

The paper is organized as follows. An overview of SCLP is given in Section 2; Section 3 briefly introduces the reader to the Gurevich's Abstract State Machine. The operational semantics of the language in terms of ASMs is presented in Section 4; conclusions and future directions are mentioned in Section 5.

*University of Pisa - Computer Science Department - Corso Italia 40, 56125 Pisa, Italy - E-mail: bista@di.unipi.it

†University of Catania - Department of Mathematics - Viale A. Doria 6, 95125 Catania, Italy - E-mail: riccobene@dipmat.unict.it

2 The SCLP framework

Constraint logic programming (CLP) [9] language extends logic programming (LP) by replacing term equalities with constraints and unification with constraint solving. Programming in CLP means choosing a constraint system for a specific class of constraints (for example, linear arithmetic constraints, or finite domain constraints) and embedding it into a logic programming engine. This approach is very flexible since one can choose among many constraint systems without changing the overall programming language, and has shown to be very successful in specifying and solving complex problems in terms of constraints of various kind. However, it can handle only classical constraint solving.

SCLP (for Semiring-based CLP) has been defined in [6] as an extension of the CLP formalism in order to handle also SCSP problems. The SCSP framework extends, in turn, the Constraint Satisfaction Problems (CSPs) formalism [8] in order to model both classical constraint solving and more sophisticated features like uncertainty, probability, fuzziness and optimization.

The base of both the SCSP and the SCLP approach is the notion of semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$. The elements of the chosen set A can be interpreted in many ways: cost, level of preference, certainty, probability, etc. The $\times$ and $+$ operations can be considered respectively as the operation of combination between constraints and the operation of choice between two admissible solutions. In fact, the structure of semiring S allows to define a partial order $\leq_S$ on the set A such that $a \leq_S b$ if $a + b = b$. We say that solution s_1 with cost a_1 is *better* than solution s_2 with cost a_2 if $a_2 \leq_S a_1$.

In passing from CLP to SCLP languages, classical constraints have been replaced with the more general SCSP constraints. By doing this, the notions of interpretation, model, model intersection, and others, have been modified since the semiring operations must be taken into account instead of the usual CLP operations. For example, while CLP interpretations associate a truth value (either *true* or *false*) to each ground atom, in the SCLP framework ground atoms yield elements of the semiring. Furthermore, while in CLP the value associated to a conjunction of atoms is the *logical and* among the truth values associated to each of its instantiations, in SCLP the *and* is replaced with the $\times$ operation of the semiring.

SCLP(S, D) programs are just CLP(FD) programs (Constraint Logic Programs on Finite Domain) [9] where constraints have a cost upon the chosen semiring S and variables can be assigned values over a finite set D. Note that a suitable extension of the SCLP(S, D) framework might handle constraints on a non finite domain D, as it happens for the CLP approach.

A program is a set of *clauses* composed by a *head* and a *body*. The head is just an atom and the body is either a collection of atoms, or a value of the semiring. A *goal* is a collection of atoms, and the *initial queries* consist of ground atoms. The BNF for this syntax follows.

$P ::= CL \mid CL, P$
$CL ::= H : -B$
$H ::= AT$ where AT is the category of atoms
$LAT ::= \Box \mid LAT, AT$
$B ::= LAT \mid a$ where $a \in A$
$G ::= \quad : -LAT$

As an example, consider the following SCLP(S, D) program where $S = \langle -N \cup \{-\infty\}, max, +, -\infty, 0 \rangle$, $-N$ is the set of non-positive integers, and $D = \{a, b, c\}$. The partial order relation $\leq_S$ is the canonical $\leq$ ordering over integers.

```
s(x)   :- p(x,y).
p(a,b) :- q(a).
p(a,c) :- r(a).
q(a)   :- t(a).
t(a)   :- -2.
r(a)   :- -3.
```

The intuitive meaning of a semiring value like -3 associated to the atom $r(a)$ is that $r(a)$ costs 3 units. Thus the set $-N \cup \{-\infty\}$ contains all possible costs, and the choice of the two operations max and $+$ implies that we intend to maximize the sum of the costs. This gives us the possibility to select the atom instantiation which gives the minimal cost overall.

In order to define the *operational semantics* of the language using the resolution rule as done for CLP, the SCLP programs must be rewritten according to the program transformation presented in [6]. The BNF syntax of the rewritten programs is the following:
$H ::= AT$ where AT is the category of atoms
$LAT ::= \Box \mid LAT, AT$
$B2 ::= \langle LAT, \theta, a\rangle$ where $a \in A$
$CL1 ::= H : -B2$
$P1 ::= CL1 \mid CL1, P1$
$G1 ::= B2$

Therefore, the previous program becomes as follows:

```
s(x)   :-   ⟨ p(x,y), ε,0⟩.
p(x,y) :-   ⟨ q(a), {x=a,y=b}, 0⟩.
p(x,y) :-   ⟨ r(a), {x=a,y=c}, 0⟩.
q(x)   :-   ⟨ t(a), {x=a}, 0⟩.
t(x)   :-   ⟨□, {x=a}, -2 ⟩.
r(x)   :-   ⟨□, {x=a}, -3 ⟩.
```

On the transformed SCLP programs, we apply the following *semantic rule*. It defines the transitions of a nondeterministic transition system whose states are goals (according to the syntactic category G1).

If the first atom of the current goal unifies with the head of a clause, then we can replace that atom with the body of the selected clause, performing a step similar to the resolution step in CLP. The main difference here is that we must update the third element of the goal representing the 'cost' over the semiring: if before the transition this value is a and the transition uses a clause whose body has value a_1, then the value associated to the new goal is $a \times a_1$. The reason for using the $\times$ operation of the semiring is that this is exactly the operation used when accumulating constraints in the SCSP framework.

$$\frac{\begin{array}{c} C = A, Cr \\ A' : -\langle C_1, \theta_1, a_1\rangle \text{ is the selected clause} \\ \exists\theta' = mgu(A\theta, A'\theta_1) \end{array}}{\langle C, \theta, a\rangle \longrightarrow \langle (C_1, Cr), \theta' \circ \theta_1, a \times a_1\rangle}$$

A *derivation* is a finite or infinite sequence of applications of the above rule. A *refutation* is a finite derivation whose *final goal* is of the form $\langle \Box, \theta, a\rangle$.

Let us now consider the set $S = \{\langle C, a\rangle \mid \langle C, \varepsilon, \mathbf{1}\rangle \Rightarrow^* \langle \Box, \theta, a\rangle \text{ for some } \theta\}$ which contains all pairs representing all refutations for the given program. Note that we can forget about the derivation θ accumulated during the refutations, since we always start with a ground goal. Thus θ only refers to variables introduced during the derivation.

Let $OS_P : AT \to A$ be the function that associates with a ground atom a value on the semiring set; $OS_P(C) = \sum_{a_i \mid \langle C, a_i\rangle \in S} a_i$ is defined as the *operational semantics* of the query C w.r.t. the program P. The symbol $\sum$ yields the semiring sum operation. Note that, if the set of all a_i such that $\langle C, a_i\rangle$ is in S is empty, $OS_P(C)$ returns the unit element $\mathbf{0}$ for $+$.

For example, by considering the goal $\langle s(a), \varepsilon, \mathbf{1}\rangle$, we get two refutations, one represented by the pair $\langle s(a), -2\rangle$, and the other one by $\langle s(a), -3\rangle$. Thus $OS_P(s(a)) = max(-2, -3) = -2$.

Remark. In the sequel we consider only SCLP programs rewritten according to the syntactic category G1.

3 The Gurevich's Abstract State Machine

We assume that the reader is familiar with the semantics of the Abstract State Machine defined in [4], and we quote here only the essential definitions.

Given a program *Prog* (consisting of a finite number of *transition rules*) and a (class of) *initial state*(s) S_0, a *Gurevich's Abstract State Machine* $\mathcal{A}$ models the operational behavior of a real dynamic system $\mathcal{S}$ in terms of state transitions. A *state* is a first-order structure representing the instantaneous configuration of $\mathcal{S}$.

The basic form of a transition rule is the following *function update*

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1} which differs from S_i only in the new interpretation of the function f.

There are some rule constructors.

- The *conditional constructor* which produces "guarded" transition rules of the form:

  $$\texttt{if } g \texttt{ then } R_1 \texttt{ else } R_2$$

  where g is a ground term (the *guard* of the rule) and R_1, R_2 are transition rules. To fire that new rule to a state S_i, $i \geq 0$, evaluate the guard; if it is true, then execute R_1, otherwise execute R_2. The `else` part may be omitted.

- The *parallel constructor* which produces the "parallel" synchronous rule of form:

  $$\texttt{var } x \texttt{ ranges over } U$$
  $$R(x)$$

  where $R(x)$ is a generalized basic transition rule with a variable x ranging over the universe U.

  The operational meaning of the constructor consists in the execution of the new rule $R(x)$ for every $x \in U$.

We also make use of the following construct to let universes grow:

$$\texttt{extend } U \texttt{ by } x_1, \ldots, x_n \texttt{ with } Updates \texttt{ endextend}$$

where *Updates* may (and should) depend on the x_i and are used to define certain functions for (some of) the new objects x_i of the resulting universe U.

State transitions of $\mathcal{A}$ may be influenced in two ways: *internally*, through the rules of the program *Prog*, or *externally* through the modifications of the environment. A non static function f that is not updated in any transition rule is called *oracle function* if its values are given non-deterministically by an oracle.

A computation of $\mathcal{S}$ is modeled through a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of $\mathcal{A}$, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the rules of *Prog* to S_n.

4 The abstract operational model of SCLP

4.1 Signature

This section explains the basic data types (domains and functions) which are used in the transition rules of Section 4.2.

A SCLP computation can be seen as parallel search for all solutions (final goals) of an initially given query, to be found in a space structured as a tree. Each query's computation is represented by a branch of the tree and the breadth search strategy is similar to the OR-Parallel model for Prolog.

The operational result of a computation is the (semiring) sum of the values of all computed solutions; since some branches might lead to infinite computations as well as to useless solutions (final goal whose semiring value a is worse than the sum of the values associated to the already computed solutions), we introduce some optimization techniques which allow to cut away all 'bad' branches, thus avoiding useless computations.

We represent the search space by a set *NODE* of nodes and the tree structure on *NODE* is formalized by a partial function $parent : NODE \rightarrow NODE$ (undefined on *root*), to be dynamically updated through rules.

All possible SCLP *computation states* intuitively correspond to the levels of the tree, therefore they are represented by the subset of *NODE* made up of all *active* nodes (see below) at that state.

Each element n of *NODE* has to carry the relevant information for a complete description – at this abstraction level – of the (sub)computation in the subtree rooted at n. This information consists of the current goal $G = \langle C, \theta, a \rangle$, where C is the collection of atoms which still have to be computed, θ is the substitution computed so far and a is the value of the semiring representing the *cost* (or level of preference, certainty, probability, etc.) accumulated performing the previous (part of the) computation.

Hence, we introduce the function

$$goal : NODE \rightarrow GOAL$$

associating with each node the goal which has still to be computed.

GOAL is the set of SCLP goals, and it yields $GOAL = AT^* \times SUB \times A$, where AT denotes the universe of atoms, *SUB* denotes the set of substitutions, A is the set of the semiring $S =< A, +, \times, \mathbf{0}, \mathbf{1} >$ (see Section 2).

GOAL comes with the following functions yielding the three components of a goal:

$$listatom : GOAL \rightarrow AT^*$$
$$sub : GOAL \rightarrow SUB$$
$$cost : GOAL \rightarrow A$$

SUB is a set of (not further specified) *substitutions* and comes together with three abstract functions

$$mgu : TERM \times TERM \rightarrow SUB \cup \{undef\}$$

associating to two terms either their most general unifier substitution or the answer that there is not any; the substitution applying function

$$apply : TERM \times SUB \rightarrow TERM$$

yielding the result of applying the given substitution to a given term; substitution concatenation

$$\circ : SUB \times SUB \rightarrow SUB$$

TERM is the set of generic SCLP terms (atoms or collections of atoms).

CLAUSE denotes the set of program statements and comes with auxiliary functions yielding *head* and *body* of SCLP clauses

$$clhead\ : CLAUSE \rightarrow AT$$
$$clbody\ : CLAUSE \rightarrow GOAL$$

The current program is represented by a distinguished element *db* (database) of a universe *PROGRAM*. The candidate clauses which have to be considered in a call to define the alternatives of a goal computation, are accessed using an abstract function

$$procdef : AT \times PROGRAM \times \mathcal{N} \rightarrow CLAUSE^*$$

which yields the (renamed) clauses defining, in the given program, the predicate having the same functor as the selected atom. $\mathcal{N}$ represents the set of natural numbers and the following function yields the current renaming index

$$lev : NODE \rightarrow \mathcal{N}$$

This function is recursively defined as follows:

$$lev(n) = \begin{cases} 0 & \text{if } n = root \\ 1 + lev(parent(n)) & \text{otherwise} \end{cases}$$

We also allow to use the functions *head* and *tail* on lists with obvious meaning, and the function $proj(i, L)$ to get the i-th element of a list L.

We use the abbreviation

$$act(n) \equiv head(listatom(goal(n)))$$

to select the current atom to compute at the level of the node n.

In order to control the computation of useless solutions, we define the subset $SUCCLEAF = \{n \in NODE \mid goal(n) = \langle \Box, \theta, a\rangle, \theta \in SUB, a \in COST\}$ of $NODE$ containing all the leaf nodes of the search tree whose paths represent successful computations already performed. We also introduce the global variable

$$estimate = \begin{cases} \sum_{n \in SUCCLEAF} cost(goal(n)) & \text{if } SUCCLEAF \neq \emptyset \\ \mathbf{0} & \text{otherwise} \end{cases}$$

which yields, at each state, the *best* semiring value among those associated to the already computed solutions. If the value a of the current goal $\langle C, \theta, a\rangle$ is worse than *estimate*, this goal computation has not to be carried on. We also make use of the predicate

$$admissible : A \rightarrow \{true, false\}$$

such that

$$admissible(c) = \begin{cases} true & \text{if } estimate \leq_S c \\ false & \text{otherwise} \end{cases}$$

where $\leq_S$ is the partial order relation on the semiring S.

Since the SCLP terms do not contain any function symbol and the initial queries consist of only ground atoms, all infinite computations can be found out by looking for cycles; a cycle occurs along a branch when there are two nodes containing the same goal (without considering variable renaming). This strategy allow us to stop all infinite computations.

Therefore, we introduce the function

$$double : NODE \rightarrow \{true, false\}$$

which is true on a given n if there is an ancestor node of n having the same sequence of atoms to compute. Formally speaking,

$$double(n) = \begin{cases} true & \text{if } \exists n' \in ancestor(n) : comp(n') \doteq [comp(n)]\theta, \text{ for some renamed subst. } \theta \\ false & \text{otherwise} \end{cases}$$

where $ancestor(n)$ yields the set of all nodes along the path from n to the *root*, $\doteq$ stands for the syntactical equality, and $comp(n) \equiv apply(listatom(goal(n)), sub(goal(n)))$.

To be able to speak about *termination* we will distinguish between *active* and *non-active* nodes by the function

$$active : NODE \rightarrow \{true, false\}$$

A node borns active and becomes non-active either after performing its own computation step or if one of the following three cases occurs:

- computation failure: when the first atom of the current goal does not unify with the head of the selected clause;
- the current goal will be an useless solution: when its semiring value is worse than *estimate*;
- infinite computation: when the current node already occurs along its path.

To keep information of all computed solutions of the initial query, we define the subset $SOL = \{g \in GOAL \mid g = \langle \Box, \theta, a\rangle\}$ of $GOAL$.

4.2 Transition rules

We now define the rules by which the system, starting from an *initial state*, tries to reach successful execution of the query $\mathtt{Q}$ ($\neq \Box$) by a given program $\mathtt{P}$. The initial state satisfies the conditions: $NODE = \{root\}$; $goal(root) = \langle \mathtt{Q}, \varepsilon, \mathbf{1}\rangle$; $active(root) = true$; $SUCCLEAF = \emptyset$; $SOL = \emptyset$.

According to the operational semantic rule of a SCLP programs defined in Section 2, the basic computation step consists of splitting the current goal $\langle (A, Cr), \theta, a\rangle$ associated with a node n, into m goals of the form $\langle (C_i, Cr), \theta'_i \circ \theta_i, a \times a_i\rangle$, where m is the number of candidate clauses for A, $A_i : -\langle C_i, \theta_i, a_i\rangle$, $1 \leq i \leq m$, is the i-th candidate clause for A and $\theta'_i = mgu(A\theta, A_i\theta_i)$. All the new m goals are computed in parallel. We can imagine the tree representing the computational space grows in breadth at each computation step performed by all the active nodes.

If an active node n has an admissible semiring value (i.e. $admissible(cost(goal(n)) \neq false)$, does not belong to an infinite branch (i.e. $double(n) \neq true$) and its computation has not failed (i.e. $sub(goal(n)) \neq undef$), by the following **Reduction Rule** as many child nodes of n are created as the candidate clauses there are for the activator $act(n)$ of n; and n becomes non-active. Otherwise the active node becomes non-active, as a leaf of the tree, without performing any splitting and stopping the computation along its path. In such a way all the "bad" branches are pruned. This strategy allows to avoid the computation of non optimal solutions and infinite computations, thus optimizing the computation of the initial query.

Each child node is created *active* and receives as goal the result of the resolution between its parent goal and the relative selected candidate clause. This resolution step is performed by the macro $set\_goal(t_i, n)$ according to the semantic rule given in Section 2.

Reduction Rule

`var` n `ranges over` $NODE$

`If` $active(n)$
`then if` $sub(goal(n)) \neq undef$
$\quad\wedge\ admissible(cost(goal(n)))$
$\quad\wedge\ \neg double(n)$
`then let` $l = length(procdef(act(n),\ db,\ lev(n)))$
`extend` NODE `by` $t_1, t_2, \ldots, t_l$ `with`
$parent(t_i) := n$
$active(t_i) := true$
$set\_goal(t_i, n)$
`endextend`
$active(n) := false$

$$set\_goal(t_i, n) \equiv \texttt{let}\ clause = proj(i,\ procdef(act(n),\ db,\ lev(n)))$$
$$goal(t_i) := \langle\ [listatom(clbody(clause)) \mid tail(listatom(goal(n))],$$
$$mgu(apply(act(n), sub(goal(n))),$$
$$apply(clhead(clause), sub(clbody(clause))),$$
$$cost(goal(n)) \times cost(clbody(clause))\rangle$$

If the tree cannot grow any more, i.e. if all the nodes are not active, by the following **Collect Rule** all the computed solutions of the initial query are collected into the set SOL.

Collect Rule

`If` $\forall$ n $\in NODE$: $\neg active(n)$
`then`
$SOL := \bigcup_{n \in SUCCLEAF} goal(n)$

According to the OS function definition given in Section 2, we can compute the operational meaning of the program `P` with initial query `Q` as $OS_{\texttt{P}}(\texttt{Q}) = \sum_{g \in SOL} cost(g)$.

5 Conclusion and future work

We have presented an abstract ASM model to specify the operational behavior of SCLP programs. This model can be viewed as the first step versus a real implementation of the language. The model faithfully reflects the operational semantics of SCLP programs, but it also suggests a possible strategy parallelizing the computation of all possible solutions. In fact, we have formalized some useful techniques which allow to cut infinite and/or non optimal paths. A good heuristic, depending on the type of the problem, could be useful to adress a faster search of an estimate via a depth first visit of the tree.

We are now investigating how to refine abstract functions and domains to efficiently design the language implementations which are reachable through a sequence of correctly refined ASMs.

References

[1] C.Beierle, E.Börger, I.Durdanivich, U.Glässer, E.Riccobene, "Refining Abstract State Machine Specifications of the Steam Boiler Control to Well Documented Executable Code", in *Formal Methods for Industrial Applications: The Steam-Boiler Case Study* (Eds. J.R.Abrial, E.Börger, H.Langmaack), Springer Verlag, LNCS 1165, pp.52-78.

[2] E.Börger, "Specification and Validation Methods", Oxford University Press (1994).

[3] E.Börger, "Why Use Evolving Algebras for Hardware and Software Engineering?"; in M. Bartosek, J. Staudek, J. Wiedermann (Eds.): Proc. SOFSEM'95, Springer LNCS 1012 (1995), 236-271.

[4] Y.Gurevich, "Evolving Algebras 1993: Lipari Guide"; in E. Börger (Ed.): Specification and Validation Methods, Oxford University Press (1995).

[5] S.Bistarelli, U.Montanari, F.Rossi, "Constraint Solving over Semirings", in *Proc. IJCAI95*, Morgan Kaufman, 1995.

[6] S.Bistarelli, U.Montanari, F.Rossi, "Semiring-based Constraint logic programming", in *Proc. IJCAI97.*

[7] S.Bistarelli, U.Montanari, F.Rossi, "Semiring-based Constraint Solving over Semiring and Optimization", *Journal of the ACM*, Vol. 44, n. 2 (March 1997), pp. 201-236.

[8] A.K.Mackworth, "Constraint Satisfaction", in Stuart C. Shapiro, editor, *Encyclopedia of AI (second edition)*, Vol. 1, pp.285-293. John Wiley & Sons, 1992.

[9] J.Jaffar, J.Lassez, "Constraint Logic Programming", in *Proc. POPL. ACM, 1987.*