Applications of Polyhedral Computations to the Analysis and Verification of Hardware and Software Systems[☆]

Roberto Bagnara[a], Patricia M. Hill[b], Enea Zaffanella[a]

[a]*Department of Mathematics, University of Parma, Italy*
[b]*School of Computing, University of Leeds, UK*

Abstract

Convex polyhedra are the basis for several abstractions used in static analysis and computer-aided verification of complex and sometimes mission critical systems. For such applications, the identification of an appropriate complexity-precision trade-off is a particularly acute problem, so that the availability of a wide spectrum of alternative solutions is mandatory. We survey the range of applications of polyhedral computations in this area; give an overview of the different classes of polyhedra that may be adopted; outline the main polyhedral operations required by automatic analyzers and verifiers; and look at some possible combinations of polyhedra with other numerical abstractions that have the potential to improve the precision of the analysis. Areas where further theoretical investigations can result in important contributions are highlighted.

Key words: Static analysis, computer-aided verification, abstract interpretation.

1. Introduction

The application of polyhedral computations to the analysis and verification of computer programs has its origin in a groundbreaking paper by Cousot and Halbwachs [31]. There, the authors applied the theory of abstract interpretation [28, 29] to the static determination of linear equality and inequality relations among program variables. In essence, the idea consists in interpreting a program (as will be explained in more detail in Sections 2.1 and 3) on a domain of convex polyhedra instead of the concrete domain of (sets of vectors of) machine numbers. Each program operation is correctly approximated by a corresponding

[☆]This work has been partly supported by PRIN project "AIDA: Abstract Interpretation Design and Applications."

Email addresses: bagnara@cs.unipr.it (Roberto Bagnara), hill@comp.leeds.ac.uk (Patricia M. Hill), zaffanella@cs.unipr.it (Enea Zaffanella) operation on polyhedra and measures are taken to ensure that the approximate computation always terminates. At the end of this process, the obtained polyhedra encode provably correct *linear invariants* of the analyzed program (i.e., linear equalities and inequalities that are guaranteed to hold for each program execution and for each program input).

As we show in this paper, relational information concerning the data objects manipulated by programs or other devices is crucial for a broad range of applications in the field of automatic or semi-automatic program manipulation: it can be used to prove the absence of certain kinds of errors; it can verify that certain processes always terminate or stabilize; it can pinpoint the position of errors in the system; and it can enable the application of optimizations. Despite this, due to the lack of efficient, robust and publicly available implementations of convex polyhedra and of the required operations, the line of work begun by Cousot and Halbwachs did not see much development until the beginning of the 1990s. Since then, this approach has been increasingly adopted and today convex polyhedra are the basis for several abstractions used in static analysis and computer-aided verification of complex and sometimes mission critical systems. For such applications, the identification of an appropriate complexity-precision trade-off is a particularly acute problem: on the one hand, relational information provided by general polyhedra is extremely valuable; on the other hand, its high computational cost makes it a fairly scarce resource that must be managed with care. This implies, among other things, that general polyhedra must be combined with simpler polyhedra in order to achieve scalability. As the complexity-precision trade-off varies considerably between different applications, the availability of a wide spectrum of alternative solutions is mandatory.

In this paper, we survey the range of applications of polyhedral computations in the area of the analysis and verification of hardware and software systems: we describe in detail one important —and historically, first— application of polyhedral computations in the field of formal methods, the linear invariant analysis for imperative programs; we provide an account of linear hybrid systems that is based directly on polyhedra; and we explain with an example how polyhedral approximations can be applied to analog systems. The paper also provides an overview of the main polyhedral operations required by these applications, brief descriptions of some of the different classes of polyhedra that may be adopted, depending on the particular context, and a look at some possible combinations of polyhedra with other numerical abstractions that have the potential to improve the precision of the analysis. Areas where further theoretical investigations can result in important contributions are highlighted. Some bibliographic references and a few examples have been omitted from this paper for space reasons; the interested reader can find them in the technical report version [13].

The plan of the paper is as follows. Section 2 introduces the required notions and notations. Section 3 demonstrates the use of polyhedral computations in the specification of a linear invariant analysis for a simple imperative language. Section 4 is devoted to polyhedral approximation techniques for hybrid systems, which, as shown in Section 5 can also be applied to purely analog systems. Section 6 presents several families of polyhedral approximations. The most important operations that such approximations must provide are illustrated in Section 7. Section 8 concludes.

2. Preliminaries

We assume some basic knowledge about lattice theory [20]. Let $(S, \sqsubseteq)$ and $(T, \preceq)$ be two partially ordered sets; the function $f \colon S \to T$ is *monotonic* if, for all $x_0, x_1 \in S$, $x_0 \sqsubseteq x_1$ implies $f(x_0) \preceq f(x_1)$. If $(S, \sqsubseteq) \equiv (T, \preceq)$, so that $f \colon S \to S$, an element $x \in S$ such that $x = f(x)$ is a *fixpoint* of f. If $(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice, then f is *continuous* if it preserves the least upper bound of all increasing chains, i.e., for all $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ in S, it satisfies $f\bigl(\bigsqcup x_i\bigr) = \bigsqcup f(x_i)$; in such a case, the least fixpoint of f with respect to the partial order '$\sqsubseteq$', denoted $\operatorname{lfp} f$, can be obtained by iterating the application of f starting from the bottom element $\bot$, thereby computing the upward iteration sequence $\bot = f^0(\bot) \sqsubseteq f^1(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots \sqsubseteq f^i(\bot) \sqsubseteq \cdots$, up to the first non-zero limit ordinal ω; namely, $\operatorname{lfp} f = f^\omega(\bot) \overset{\mathrm{def}}{=} \bigsqcup_{i<\omega} f^i(\bot)$.

For each $f_0 \colon S_0 \to T_0$ and $f_1 \colon S_1 \to T_1$, the function $f_0[f_1] \colon (S_0 \cup S_1) \to (T_0 \cup T_1)$ is defined, for each $x \in S_0 \cup S_1$, so that $f_0[f_1](x) = f_1(x)$, if $x \in S_1$, and $f_0[f_1](x) = f_0(x)$, otherwise.

For $n > 0$, we denote by $\mathbf{v} = (v_0, \ldots, v_{n-1}) \in \mathbb{R}^n$ an n-tuple (vector) of real numbers; $\mathbb{R}_+$ is the set of non-negative real numbers; $\langle \mathbf{v}, \mathbf{w} \rangle$ denotes the scalar product of vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$; the vector $\mathbf{0} \in \mathbb{R}^n$ has all components equal to zero. We write $\mathbf{v} :: \mathbf{w}$ to denote the *tuple concatenation* of $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^m$, so that $\mathbf{v} :: \mathbf{w} \in \mathbb{R}^{n+m}$.

Let $\mathbf{x}$ be an n-tuple of distinct variables. Then $\beta = \bigl(\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b\bigr)$ denotes a linear inequality constraint, for each vector $\mathbf{a} \in \mathbb{R}^n$, where $\mathbf{a} \neq \mathbf{0}$, each scalar $b \in \mathbb{R}$, and $\bowtie \in \{\geq, >\}$. A linear inequality constraint β defines a (topologically closed or open) affine half-space of $\mathbb{R}^n$, denoted by $\operatorname{con}\bigl(\{\beta\}\bigr)$.

A set $\mathcal{P} \subseteq \mathbb{R}^n$ is a *(convex) polyhedron* if and only if $\mathcal{P}$ can be expressed as the intersection of a finite number of affine half-spaces of $\mathbb{R}^n$, i.e., as the solution $\mathcal{P} = \operatorname{con}(\mathcal{C})$ of a finite set of linear inequality constraints $\mathcal{C}$ (called a *constraint system*). The set of all polyhedra on the vector space $\mathbb{R}^n$ is denoted as $\mathbb{P}_n$. When partially ordered by set-inclusion, convex polyhedra form a lattice $(\mathbb{P}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap)$ having the empty set and $\mathbb{R}^n$ as the bottom and top elements, respectively; the binary meet operation, returning the greatest polyhedron smaller than or equal to the two arguments, is easily seen to correspond to set-intersection; the binary join operation, returning the least polyhedron greater than or equal to the two arguments, is denoted '$\uplus$' and called *convex polyhedral hull* (poly-hull, for short). In general, the poly-hull of two polyhedra is different from their convex hull [76].

A relation $\psi \subseteq \mathbb{R}^n \times \mathbb{R}^n$ (of dimension n) is said to be *affine* if there exists $\ell \in \mathbb{N}$ and $\mathbf{a}_i, \mathbf{c}_i \in \mathbb{R}^n$, $b_i \in \mathbb{R}$ and $\bowtie_i \in \{\geq, >\}$, for each $i = 1, \ldots, \ell$, such that

$$\forall \mathbf{v}, \mathbf{w} \in \mathbb{R}^n : (\mathbf{v}, \mathbf{w}) \in \psi \iff \bigwedge_{i=1}^{\ell} \bigl(\langle \mathbf{c}_i, \mathbf{w} \rangle \bowtie_i \langle \mathbf{a}_i, \mathbf{v} \rangle + b_i\bigr).$$

Any affine relation of dimension n can thus be encoded by ℓ linear inequalities on a $2n$-tuple of distinct variables $\mathbf{x} :: \mathbf{x}'$ (playing the role of $\mathbf{v}$ and $\mathbf{w}$, respectively), therefore defining a polyhedron in $\mathbb{P}_{2n}$. The set of polyhedra $\mathbb{P}_n$ is closed under the (direct or inverse) application of affine relations: i.e., for each $\mathcal{P} \in \mathbb{P}_n$ and each affine relation $\psi \subseteq \mathbb{R}^n \times \mathbb{R}^n$, the image $\psi(\mathcal{P})$ and the preimage $\psi^{-1}(\mathcal{P})$ are in $\mathbb{P}_n$.

2.1. Abstract Interpretation

The semantics of a hardware or software system is a mathematical description of all its possible run-time behaviors. Different semantics can be defined for the same system, depending on the details being recorded. *Abstract interpretation* [28, 29] is a formal method for relating these semantics according to their level of abstraction, so that questions about the behavior of a system can be provided with sound, possibly approximate answers.

The concrete semantics $c \in C$ of a program is usually formalized as the least fixpoint of a continuous semantic function $\mathcal{F} \colon C \to C$, where the concrete domain $(C, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice of semantic properties; in many interesting cases, the computational order '$\sqsubseteq$' corresponds to the approximation relation, so that $c_1 \sqsubseteq c_2$ holds if c_1 is a stronger property than c_2 (i.e., c_2 correctly approximates c_1).

For instance, the run-time behavior of a program may be defined in terms of a transition system $\langle \Sigma, t, \iota \rangle$, where Σ is a set of states, $\iota \subseteq \Sigma$ is the subset of initial states, and $t \in \wp(\Sigma \times \Sigma)$ is a binary transition relation mapping a state to its possible successor states. Letting $\Sigma^\star$ denote the set of all finite sequences of elements in Σ, the initial history of a forward computation can be recorded as a partial execution trace $\tau = \sigma_0 \cdots \sigma_m \in \Sigma^\star$ starting from an initial state $\sigma_0 \in \iota$ and such that any two consecutive states σ_i and σ_{i+1} are related by the transition relation, i.e., $(\sigma_i, \sigma_{i+1}) \in t$. In such a context, an element of the concrete domain $\bigl(\wp(\Sigma^\star), \subseteq, \emptyset, \Sigma^\star, \cup, \cap\bigr)$ is a set of partial execution traces and the concrete semantics is $\mathrm{lfp}(\mathcal{F})$, where the semantic function is defined by

$$\begin{aligned}\mathcal{F} = \lambda X \in \wp(\Sigma^\star) \,.\, X \cup \{\, \tau \in \Sigma^\star \mid \tau = \sigma_0 \in \iota \,\} \\ \cup \bigl\{\, \tau\sigma_{i+1} \in \Sigma^\star \bigm| \tau = \sigma_0 \cdots \sigma_i \in X, (\sigma_i, \sigma_{i+1}) \in t \,\bigr\}.\end{aligned}$$

An abstract domain[1] $(D^\sharp, \sqsubseteq, \bot, \sqcup)$ can be often modeled as a bounded join-semilattice, so that it has a bottom element $\bot$ and the least upper bound $d_1^\sharp \sqcup d_2^\sharp$ exists for all $d_1^\sharp, d_2^\sharp \in D^\sharp$. This domain is related to the concrete domain by a monotonic and injective concretization function $\gamma \colon D^\sharp \to C$. Monotonicity and injectivity mean that the abstract partial order is equivalent to the approximation relation induced on $D^\sharp$ by the concretization function γ. Conversely, the concrete domain is related to the abstract one by a partial abstraction function $\alpha \colon C \rightarrowtail D^\sharp$ such that, for each $c \in C$, if $\alpha(c)$ is defined then $c \sqsubseteq \gamma\bigl(\alpha(c)\bigr)$. In

[1] To avoid notational burden, we will freely overload the lattice-theoretic symbols '$\sqsubseteq$', '$\bot$', '$\sqcup$', etc., exploiting context to disambiguate their meaning.

particular, we assume that $\alpha(\bot) = \bot$ is always defined; when needed or useful, we will require a few additional properties.

For example, a first abstraction of the semantics above, typically adopted for the inference of invariance properties of programs [28, 29], approximates a set of traces by the set of states occurring in any one of the traces. The *reachable states* are thus characterized by elements of the complete lattice $\bigl(\wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap\bigr)$, which plays here the role of the abstract domain. The concretization function relating $D^\sharp = \wp(\Sigma)$ to $C = \wp(\Sigma^\star)$ is defined, for each $d^\sharp \in \wp(\Sigma)$, by $\gamma(d^\sharp) \stackrel{\mathrm{def}}{=} \{\, \tau \in \Sigma^\star \mid \tau = \sigma_0 \cdots \sigma_m, \forall i = 0, \ldots, m : \sigma_i \in d^\sharp \,\}$. The concrete semantic function $\mathcal{F} \colon \wp(\Sigma^\star) \to \wp(\Sigma^\star)$ can thus be approximated by the monotonic abstract semantic function $\mathcal{A} \colon \wp(\Sigma) \to \wp(\Sigma)$ defined by

$$\mathcal{A} = \lambda d^\sharp \in \wp(\Sigma) \mathrel{.} d^\sharp \cup \iota \cup \bigl\{\, \sigma' \in \Sigma \bigm| \exists \sigma \in d^\sharp \mathrel{.} (\sigma, \sigma') \in t \,\bigr\}.$$

This abstract semantic function is *sound* with respect to the concrete semantic function in that it satisfies the local correctness requirement

$$\forall c \in C : \forall d^\sharp \in D^\sharp : c \sqsubseteq \gamma(d^\sharp) \implies \mathcal{F}(c) \sqsubseteq \gamma\bigl(\mathcal{A}(d^\sharp)\bigr),$$

ensuring that each iteration $\mathcal{F}^i(\bot)$ in the concrete fixpoint computation is approximated by computing the corresponding abstract iteration $\mathcal{A}^i\bigl(\alpha(\bot)\bigr)$. In particular, the least fixpoint of $\mathcal{F}$ is approximated by any post-fixpoint of $\mathcal{A}$ [29], i.e., any abstract element $d^\sharp \in D^\sharp$ such that $\mathcal{A}(d^\sharp) \sqsubseteq d^\sharp$.

Actually, the abstraction defined above satisfies an even stronger property, in that the abstract semantic function $\mathcal{A}$ is the *most precise* of all the sound approximations of $\mathcal{F}$ that could be defined on the considered abstract domain. This happens because the two domains are related by a Galois connection [28], i.e., there exists a total abstraction function $\alpha \colon C \to D^\sharp$ satisfying

$$\forall c \in C : \forall d^\sharp \in D^\sharp : \alpha(c) \sqsubseteq d^\sharp \iff c \sqsubseteq \gamma(d^\sharp).$$

Namely, $\alpha(c) \stackrel{\mathrm{def}}{=} \bigl\{\, \sigma_i \in \Sigma \bigm| \tau = \sigma_0 \cdots \sigma_m \in c, i \in \{0, \ldots, m\} \,\bigr\}$.

For Galois connections it can be shown that $\alpha(c)$ is the best possible approximation in $D^\sharp$ for the concrete element $c \in C$; similarly, $\alpha \circ \mathcal{F} \circ \gamma$ (i.e., the function $\mathcal{A}$ defined above) is the best possible approximation for $\mathcal{F}$ [28]. Such a result is provided with a quite intuitive reading; in order to approximate the concrete function $\mathcal{F}$ on an abstract element $d^\sharp \in D^\sharp$: we first apply the concretization function γ so as to obtain the meaning of $d^\sharp$; then we apply the concrete function $\mathcal{F}$; finally, we abstract the result so as to obtain back an element of $D^\sharp$.

Abstract interpretation theory can thus be used to specify (semi-) automatic program analysis tools that are correct by design. Of course —due to well-known undecidability results— any fully automatic tool can only provide partial, though safe answers.

2.2. Abstract Domains for Numeric and Boolean Values

The reachable state abstraction described above is just one of the possible semantic approximations that can be adopted when specifying an abstract semantics. A further, typical approximation concerns the description of the states of the transition system. Each state $\sigma \in \Sigma$ may be decomposed into, e.g., a set of numerical or Boolean variables that are of interest for the application at hand; new abstract domains can be defined (and composed [28]) so as to soundly describe the possible values of these variables.

As an expository example, assume that part of a state is characterized by the value of an integer variable. Then, the domain $\bigl(\wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap\bigr)$ can be abstracted to the concrete domain of integers $\bigl(\wp(\mathrm{Int}), \subseteq, \emptyset, \mathrm{Int}, \cup, \cap\bigr)$. This domain is further approximated by an abstract domain $\bigl(\mathrm{Int}^\sharp, \sqsubseteq, \bot, \sqcup\bigr)$, via the concretization function $\gamma_\mathrm{I} \colon \mathrm{Int}^\sharp \rightarrow \wp(\mathrm{Int})$. Elements of $\mathrm{Int}^\sharp$ are denoted by $m^\sharp$, possibly subscripted. We assume that the partial abstraction function $\alpha_\mathrm{I} \colon \wp(\mathrm{Int}) \rightarrowtail \mathrm{Int}^\sharp$ is defined on all singletons $\{m\} \in \wp(\mathrm{Int})$ and on the whole set Int. We also assume that there are abstract binary operations '$\oplus$', '$\ominus$' and '$\circledast$' on $\mathrm{Int}^\sharp$ that are sound with respect to the corresponding operations on $\wp(\mathrm{Int})$ which, in turn, are the obvious pointwise extensions of addition, subtraction and multiplication over the integers. More formally, for '$\oplus$' we require $\gamma_\mathrm{I}(m_0^\sharp \oplus m_1^\sharp) \supseteq \bigl\{\, m_0 + m_1 \bigm| m_0 \in \gamma_\mathrm{I}(m_0^\sharp), m_1 \in \gamma_\mathrm{I}(m_1^\sharp) \,\bigr\}$ for each $m_0^\sharp, m_1^\sharp \in \mathrm{Int}^\sharp$, i.e., soundness with respect to addition. Similar requirements are imposed on '$\ominus$' and '$\circledast$'. Even though the definition of $\mathrm{Int}^\sharp$ is completely general, families of integer intervals come naturally to mind for this role.

Suppose now that some other part of the state is characterized by the value of a Boolean expression. Then, the domain $\bigl(\wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap\bigr)$ can be abstracted to the finite domain $\bigl(\wp(\mathrm{Bool}), \subseteq, \emptyset, \mathrm{Bool}, \cup, \cap\bigr)$, where $\mathrm{Bool} = \{\mathrm{ff}, \mathrm{tt}\}$ is the set of Boolean values. In general, such a finite domain may be further approximated by an abstract domain $(\mathrm{Bool}^\sharp, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$, related to the concrete domain by a Galois connection. Elements of $\mathrm{Bool}^\sharp$ are denoted by $t^\sharp$, possibly subscripted, and we can define abstract operations '$\obslash$', '$\ovee$' and '$\owedge$' on $\mathrm{Bool}^\sharp$ that are sound with respect to the pointwise extensions of Boolean negation, disjunction and conjunction over $\wp(\mathrm{Bool})$. For instance, for the operation '$\ovee$' to be sound with respect to disjunction on $\wp(\mathrm{Bool})$, it is required that

$$\gamma_\mathrm{B}(t_0^\sharp \ovee t_1^\sharp) \supseteq \bigl\{\, t_0 \vee t_1 \bigm| t_0 \in \gamma_\mathrm{B}(t_0^\sharp), t_1 \in \gamma_\mathrm{B}(t_1^\sharp) \,\bigr\}$$

for each $t_0^\sharp$ and $t_1^\sharp$ in $\mathrm{Bool}^\sharp$. Likewise for '$\owedge$'. For '$\obslash$' the correctness requirement is that, for each $t^\sharp$ in $\mathrm{Bool}^\sharp$, $\gamma_\mathrm{B}(\obslash t^\sharp) \supseteq \bigl\{\, \neg t \bigm| t \in \gamma_\mathrm{B}(t^\sharp) \,\bigr\}$. Abstract comparison operations $\circledcirc, \olessthan \colon \mathrm{Int}^\sharp \times \mathrm{Int}^\sharp \rightarrow \mathrm{Bool}^\sharp$ can then be defined to correctly approximate the equal-to and less-than tests: for each $m_0^\sharp, m_1^\sharp \in \mathrm{Int}^\sharp$, $\gamma_\mathrm{B}(m_0^\sharp \circledcirc m_1^\sharp) \supseteq \bigl\{\, m_0 = m_1 \bigm| m_0 \in \gamma_\mathrm{I}(m_0^\sharp), m_1 \in \gamma_\mathrm{I}(m_1^\sharp) \,\bigr\}$; likewise for '$\olessthan$'.

Simple abstract domains such as the ones above can be combined in different ways so as to obtain quite accurate approximations [28]. In some cases, however, the required precision level may only be obtained by a suitable initial choice of the abstract domain. As a notable example, suppose that some part of the state $\sigma \in \Sigma$ is characterized by n (integer or real valued) numeric variables and the application at hand needs some *relational* information about these variables. In such a context, an approximation based on a simple conjunctive combination of

n copies of the domain $\mathrm{Int}^\sharp$ described above will be almost useless. Rather, a new approximation scheme can be devised by modeling states using the domain $\bigl(\wp(\mathbb{R}^n), \subseteq, \emptyset, \mathbb{R}^n, \cup, \cap\bigr)$, where each vector $\mathbf{v} \in \mathbb{R}^n$ is meant to describe a possible valuation for the n variables. A further abstraction should map this domain so as to retain some of the relations holding between the values of the n variables. If a finite set of linear inequalities provides a good enough approximation, then the natural choice is to abstract this domain into the abstract domain of convex polyhedra $(\mathbb{P}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap)$ [31]. In this case, the concrete and abstract domains are not related by a Galois connection and, hence, a best approximation might not exist.[2] Nonetheless, the convex polyhedral hull (partial) abstraction function $\uplus \colon \wp(\mathbb{R}^n) \rightarrowtail \mathbb{P}_n$ is defined in most of the cases of interest and provides the best possible approximation. Most of the arithmetic operations seen before can be encoded (or approximated) by computing images of affine relations.

2.3. Widening Operators

It should be stressed that, in general, the abstract semantics just described is not finitely computable. For instance, both the domain of convex polyhedra and the domain of integer intervals have infinite ascending chains, so that the limit of a converging fixpoint computation cannot generally be reached in a finite number of iterations.

A finite computation can be enforced by further approximations resulting in a *Noetherian* abstract domain, i.e., a domain where all ascending chains are finite. Alternatively, and more generally, it is possible to keep an abstract domain with infinite chains, while enforcing that these chains are traversed in a finite number of iteration steps. In both cases, termination is usually achieved to the detriment of precision, so that an appropriate trade-off should be pursued. *Widening operators* [27, 29] provide a simple and general characterization for the second option.

Definition2.1. The partial operator $\nabla \colon D^\sharp \times D^\sharp \rightarrowtail D^\sharp$ is a *widening* if:

1. for all $d^\sharp, e^\sharp \in D^\sharp$, $d^\sharp \sqsubseteq e^\sharp$ implies that $d^\sharp \nabla e^\sharp$ is defined and $e^\sharp \sqsubseteq d^\sharp \nabla e^\sharp$;
2. for all increasing chains $e_0^\sharp \sqsubseteq e_1^\sharp \sqsubseteq \cdots$, the increasing chain defined by $d_0^\sharp \stackrel{\mathrm{def}}{=} e_0^\sharp$ and $d_{i+1}^\sharp \stackrel{\mathrm{def}}{=} d_i^\sharp \nabla (d_i^\sharp \sqcup e_{i+1}^\sharp)$, for $i \in \mathbb{N}$, is not strictly increasing.

It can be proved that, for any monotonic operator $\mathcal{A} \colon D^\sharp \to D^\sharp$, the upward iteration sequence with widenings starting at the bottom element $d_0^\sharp \stackrel{\mathrm{def}}{=} \bot$ and defined by

$$d_{i+1}^\sharp \stackrel{\mathrm{def}}{=} \begin{cases} d_i^\sharp, & \text{if } \mathcal{A}(d_i^\sharp) \sqsubseteq d_i^\sharp, \\ d_i^\sharp \nabla \bigl(d_i^\sharp \sqcup \mathcal{A}(d_i^\sharp)\bigr), & \text{otherwise,} \end{cases}$$

converges to a post-fixpoint of $\mathcal{A}$ after a finite number of iterations. Clearly, the choice of the widening has a deep impact on the precision of the results

[2]This happens, for instance, when approximating an n-dimensional ball with a convex polyhedron.

$$
\begin{aligned}
&m \in \mathrm{Int} \overset{\mathrm{def}}{=} \mathbb{Z} \qquad t \in \mathrm{Bool} \overset{\mathrm{def}}{=} \{\mathrm{tt}, \mathrm{ff}\} \qquad x \in \mathrm{Var} \overset{\mathrm{def}}{=} \{x_0, x_1, x_2, \ldots\}\\
&\mathrm{Aexp} \ni a ::= m \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1\\
&\mathrm{Bexp} \ni b ::= t \mid a_0 = a_1 \mid a_0 < a_1\\
&\mathrm{Stmt} \ni s ::= \mathbf{skip} \mid x := a \mid s_0; s_1 \mid \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1 \mid \mathbf{while}\, b\, \mathbf{do}\, s
\end{aligned}
$$

Figure 1: Abstract syntax of the simple imperative language

obtained. Designing a widening which is appropriate for a given application is therefore a difficult (but possibly rewarding) activity.

3. Analysis and Verification of Computer Programs

In this section we begin a review of the applications of polyhedral computations to analysis and verification problems starting with the the work of Cousot and Halbwachs [31]. This seminal paper on the automatic inference of linear invariants for imperative programs constituted a major leap forward for at least two reasons. First, the polyhedral domain proposed by Cousot and Halbwachs was considerably more powerful than all the data-flow analyses known at that time, including the rather sophisticated one by Karr which was limited to linear equalities [55]. Secondly, the use of convex polyhedra as an abstract domain established abstract interpretation as the right methodology for the definition of complex and correct program analyzers.

We illustrate the basic ideas by partially specifying the analysis of linear invariants for a very simple imperative language. The simplicity of the language we have chosen for expository purposes should not mislead the reader: the approach is generalizable to any imperative (and, for that matter, functional and logic) language [9]. The abstract syntax of the language is presented in Figure 1. The basic syntactic categories, corresponding to the sets Int, Bool and Var, are defined directly. From these, the categories of arithmetic and Boolean expressions and of statements are defined by means of BNF rules. Notice the use of syntactic meta-variables: for instance, to save typing we will consistently denote by s, possibly subscripted or superscripted, any element of Stmt.

The concrete semantics of programs is formally defined using the *natural semantics* approach [54]. This, in turn, is a "big-step" operational semantics defined by structural induction on program structures in the style of Plotkin [66]. First we define the notion of *store*, which is any mapping between a finite set of variables and elements of Int. Formally, a store is an element of the set $\mathrm{Store} \overset{\mathrm{def}}{=} \{\, \sigma\colon V \to \mathrm{Int} \mid V \subseteq \mathrm{Var},\ V \text{ finite} \,\}$ and denoted by the letter σ, possibly subscripted or superscripted. The store obtained from $\sigma \in \mathrm{Store}$ by the assignment of $m \in \mathrm{Int}$ to $x \in \mathrm{dom}(\sigma)$, denoted by $\sigma[m/x]$, is defined so that, for each $x' \in \mathrm{dom}(\sigma)$, $\sigma[m/x](x') = m$, if $x' = x$, and $\sigma[m/x](x') = \sigma(x')$, otherwise.

$$\frac{}{\langle m,\sigma\rangle \xrightarrow{\mathrm{a}} m} \qquad \frac{}{\langle x,\sigma\rangle \xrightarrow{\mathrm{a}} \sigma(x)} \qquad \frac{\langle a_0,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \quad \langle a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_1}{\langle a_0 + a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 + m_1}$$

$$\frac{\langle a_0,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \quad \langle a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_1}{\langle a_0 - a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 - m_1} \qquad \frac{\langle a_0,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \quad \langle a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_1}{\langle a_0 * a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \cdot m_1}$$

$$\frac{\langle a_0,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \quad \langle a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_1}{\langle a_0 = a_1,\sigma\rangle \xrightarrow{\mathrm{b}} (m_0 = m_1)} \qquad \frac{\langle a_0,\sigma\rangle \xrightarrow{\mathrm{a}} m_0 \quad \langle a_1,\sigma\rangle \xrightarrow{\mathrm{a}} m_1}{\langle a_0 < a_1,\sigma\rangle \xrightarrow{\mathrm{b}} (m_0 < m_1)}$$

$$\frac{}{\langle \mathbf{skip},\sigma\rangle \xrightarrow{\mathrm{s}} \sigma} \qquad \frac{\langle a,\sigma\rangle \xrightarrow{\mathrm{a}} m}{\langle x := a,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma[m/x]} \qquad \frac{\langle s_0,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'' \quad \langle s_1,\sigma''\rangle \xrightarrow{\mathrm{s}} \sigma'}{\langle s_0; s_1,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'}$$

$$\frac{\langle b,\sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{tt} \quad \langle s_0,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'} \qquad \frac{\langle b,\sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{ff} \quad \langle s_1,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'}$$

$$\frac{\langle b,\sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{ff}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma} \qquad \frac{\langle b,\sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{tt} \quad \langle c,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'' \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma''\rangle \xrightarrow{\mathrm{s}} \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'}$$

Figure 2: Concrete semantics rule schemata for the finite computations of the simple imperative language

The concrete evaluation relations that complete the definition of the concrete semantics for our simple language are defined by structural induction from a set of rule schemata. The evaluation relations for terminating computations are $\xrightarrow{\mathrm{a}} \subseteq (\mathrm{Aexp}\times\mathrm{Store})\times\mathrm{Int}$, for arithmetic expressions, $\xrightarrow{\mathrm{b}} \subseteq (\mathrm{Bexp}\times\mathrm{Store})\times\mathrm{Bool}$, for Boolean expressions, and $\xrightarrow{\mathrm{s}} \subseteq (\mathrm{Stmt}\times\mathrm{Store})\times\mathrm{Store}$, for statements. The judgment $\langle a,\sigma\rangle \xrightarrow{\mathrm{a}} m$ means that when expression a is executed in store σ it results in the integer m. The judgment $\langle b,\sigma\rangle \xrightarrow{\mathrm{b}} t$ is similar. Note that expressions do not have, in our simple language, side effects. The judgment $\langle s,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma'$ means that the statement s, executed in store σ, results in a (possibly modified) store σ'. The rule schemata, in the form $\frac{\text{premise}}{\text{conclusion}}$, that define these relations are given in Figure 2. Rule instances can be composed in the obvious way to form finite tree structures, representing finite computations.

The possibly infinite set of all finite trees is obtained by means of a least fixpoint computation, corresponding to the classical inductive interpretation of the rules in Figure 2. The rule schemata in Figure 3 can be used to directly model non-terminating computations and need to be interpreted coinductively [30]. The judgment $\langle s,\sigma\rangle \xrightarrow{\infty}$ means that the statement s diverges when executed in store σ. By a suitable adaptation of the computational ordering, both sets of finite and infinite trees can be jointly computed in a single least fixpoint

$$\frac{\langle s_0, \sigma\rangle \overset{\infty}{\rightarrow}}{\langle s_0; s_1, \sigma\rangle \overset{\infty}{\rightarrow}} \qquad \frac{\langle s_0, \sigma\rangle \xrightarrow{\mathrm{s}} \sigma' \quad \langle s_1, \sigma'\rangle \overset{\infty}{\rightarrow}}{\langle s_0; s_1, \sigma\rangle \overset{\infty}{\rightarrow}}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{tt} \quad \langle s_0, \sigma\rangle \overset{\infty}{\rightarrow}}{\langle \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \overset{\infty}{\rightarrow}} \qquad \frac{\langle b, \sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{ff} \quad \langle s_1, \sigma\rangle \overset{\infty}{\rightarrow}}{\langle \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \overset{\infty}{\rightarrow}}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{tt} \quad \langle c, \sigma\rangle \overset{\infty}{\rightarrow}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \overset{\infty}{\rightarrow}} \quad \frac{\langle b, \sigma\rangle \xrightarrow{\mathrm{b}} \mathrm{tt} \quad \langle c, \sigma\rangle \xrightarrow{\mathrm{s}} \sigma' \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma'\rangle \overset{\infty}{\rightarrow}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \overset{\infty}{\rightarrow}}$$

Figure 3: Additional concrete semantics rule schemata for the infinite computations of the simple imperative language

computation [30]. While these semantics characterizations contain all the information we need to perform a wide range of program reasoning tasks, they are generally not computable: we have thus to resort to approximation.

Following the abstract interpretation approach, as instantiated in [70], the concrete rule schemata are paired with abstract rule schemata that correctly approximate them. Before doing that, we need to formalize abstract domains for each concrete domain used by the concrete semantics.

For simple approximations of integers and Boolean expressions, we consider the abstract domains $\mathrm{Int}^\sharp$ and $\mathrm{Bool}^\sharp$ introduced in Section 2.2. The last (and most interesting) abstraction we need is one that approximates sets of stores. We thus require an abstract domain $\bigl(\mathrm{Store}^\sharp, \sqsubseteq, \bot, \sqcup\bigr)$ that is related, by means of a concretization function γ_S such that $\gamma_\mathrm{S}(\bot) = \emptyset$, to the concrete domain $\bigl(\wp(\mathrm{Store}), \subseteq, \emptyset, \mathrm{Store}, \cup, \cap\bigr)$. Elements of $\mathrm{Store}^\sharp$ are denoted by $\sigma^\sharp$, possibly subscripted. The abstract store evaluation and update operators

$$\begin{aligned}
\cdot[\cdot] &\colon (\mathrm{Store}^\sharp \times \mathrm{Aexp}) \to \mathrm{Int}^\sharp, \\
\cdot[\cdot := \cdot] &\colon \bigl(\mathrm{Store}^\sharp \times \mathrm{Var} \times \mathrm{Aexp}\bigr) \to \mathrm{Store}^\sharp, \\
\cdot[\cdot/\cdot] &\colon \bigl(\mathrm{Store}^\sharp \times \mathrm{Var} \times \mathrm{Int}^\sharp\bigr) \to \mathrm{Store}^\sharp
\end{aligned}$$

are assumed to be sound with respect to their concrete counterparts, i.e., such that, for each $\sigma^\sharp \in \mathrm{Store}^\sharp$, $a \in \mathrm{Aexp}$, $x \in \mathrm{Var}$ and $m^\sharp \in \mathrm{Int}^\sharp$:

$$\begin{aligned}
\gamma_\mathrm{I}\bigl(\sigma^\sharp[a]\bigr) &\supseteq \bigl\{\, m \in \mathrm{Int} \bigm| \sigma \in \gamma_\mathrm{S}(\sigma^\sharp), \langle a, \sigma\rangle \xrightarrow{\mathrm{a}} m \,\bigr\}, \\
\gamma_\mathrm{S}\bigl(\sigma^\sharp[x := a]\bigr) &\supseteq \bigl\{\, \sigma' \in \mathrm{Store} \bigm| \sigma \in \gamma_\mathrm{S}(\sigma^\sharp), \langle x := a, \sigma\rangle \xrightarrow{\mathrm{s}} \sigma' \,\bigr\}, \\
\gamma_\mathrm{S}\bigl(\sigma^\sharp[m^\sharp/x]\bigr) &\supseteq \bigl\{\, \sigma[m/x] \in \mathrm{Store} \bigm| \sigma \in \gamma_\mathrm{S}(\sigma^\sharp), m \in \gamma_\mathrm{I}(m^\sharp) \,\bigr\}.
\end{aligned}$$

We also need computable "Boolean filters" to refine the information contained in abstract stores, i.e., two functions $\phi_\mathrm{tt}, \phi_\mathrm{ff} \colon \mathrm{Store}^\sharp \times \mathrm{Bexp} \to \mathrm{Store}^\sharp$ such that, for each $t \in \mathrm{Bool}$, $\sigma^\sharp \in \mathrm{Store}^\sharp$ and $b \in \mathrm{Bexp}$:

$$\gamma_\mathrm{S}\bigl(\phi_t(\sigma^\sharp, b)\bigr) \supseteq \bigl\{\, \sigma \in \gamma_\mathrm{S}(\sigma^\sharp) \bigm| \langle b, \sigma\rangle \xrightarrow{\mathrm{b}} t \,\bigr\}.$$

We are now in a position to present, in Figure 4, a possible set of domain-independent abstract rule schemata. These schemata allow for the free approximation of the '$\rightsquigarrow$' right-hand sides in the conclusions. This means that if, e.g., $\frac{\text{premise}}{\langle s,\sigma\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_1^\sharp}$ is an instance of some rule, then $\frac{\text{premise}}{\langle s,\sigma\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_2^\sharp}$ is also an instance of the same rule for each $\sigma_2^\sharp$ such that $\sigma_1^\sharp \sqsubseteq \sigma_2^\sharp$. Hence the schemata in Figure 4 ensure correctness yet leaving complete freedom about precision. The ability to give up some precision, as we will see, is crucial in order to ensure the (reasonably quick) termination of the analysis.

It is possible to prove that, for each (possibly infinite) concrete tree T built using the schemata of Figures 2 and 3, for each (possibly infinite) abstract tree $T^\sharp$ built using the schemata of Figure 4, if the concrete tree root is of the form $\langle s,\sigma\rangle \xrightarrow{\mathrm{s}} \sigma_1$ (when the tree is finite) or $\langle s,\sigma\rangle \overset{\infty}{\rightarrow}$ (when the tree is infinite) and the abstract tree root is of the form $\langle s,\sigma^\sharp\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_1^\sharp$ with $\sigma \in \gamma_{\mathrm{S}}(\sigma^\sharp)$, then $T^\sharp$ correctly approximates T. This means not only that $\sigma_1 \in \gamma_{\mathrm{S}}(\sigma_1^\sharp)$ (when T is finite), but also that *each* node in T is correctly approximated by at least one node in $T^\sharp$. In other words, the abstract tree correctly approximates the entire concrete computation (see [9] for the details).

It is worth stressing the observation by Schmidt that, even when disregarding the non-terminating concrete computations, the abstract rules still have to be interpreted coinductively because most of the finite concrete trees can only be approximated by infinite abstract trees; for instance, all abstract trees containing a while loop are infinite. Since, in general, we cannot effectively compute infinite abstract trees, we still do not have a viable analysis technique. The solution is to restrict ourselves to the class of rational trees, i.e., trees with only finitely many subtrees and that, consequently, admit a finite representation.

The analysis algorithm is sketched in [70]. For expository purposes, we describe here a simplified version that, however, is enough to handle the considered programming language features. The algorithm works by recursively constructing a finite approximation for the (possibly infinite) abstract subtree rooted in the current node (initially, the root of the whole tree). The current node $n = \bigl(\langle p, \sigma_n^\sharp\rangle \rightsquigarrow r_n\bigr)$, where r_n is a placeholder for the "yet to be computed" conclusion, is processed according to the following alternatives:

1. If no ancestor of n has p in the label, the node has to be expanded using an applicable abstract rule instance. Namely, descendants of the premises of the rule are (recursively) processed, one at a time and from left to right. When the expansion of all the premises has been completed, including the case when the rule has no premise at all, the marker r_n is replaced by an abstract value computed according to the conclusion of the rule.
2. If there exists an ancestor node $m = \langle p, \sigma_m^\sharp\rangle \rightsquigarrow r_m$ of n labeled by the same syntax p and such that $\sigma_n^\sharp \sqsubseteq \sigma_m^\sharp$, i.e., if node n is subsumed by node m, then the node is not expanded further and the placeholder r_n is replaced by the least fixpoint of the equation $r_n = f_m(r_n)$, where f_m is the expression corresponding to the conclusion of the abstract rule that was used for the

$$\frac{}{\langle m, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} \alpha_{\mathrm{I}}\bigl(\{m\}\bigr)} \qquad \frac{}{\langle x, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} \sigma^\sharp[x]} \qquad \frac{\langle a_0, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \quad \langle a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_1^\sharp}{\langle a_0 + a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \oplus m_1^\sharp}$$

$$\frac{\langle a_0, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \quad \langle a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_1^\sharp}{\langle a_0 - a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \ominus m_1^\sharp} \qquad \frac{\langle a_0, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \quad \langle a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_1^\sharp}{\langle a_0 * a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \circledast m_1^\sharp}$$

$$\frac{}{\langle t, \sigma^\sharp \rangle \overset{\mathrm{b}}{\rightsquigarrow} \alpha_{\mathrm{B}}\bigl(\{t\}\bigr)} \qquad \frac{\langle a_0, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \quad \langle a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_1^\sharp}{\langle a_0 = a_1, \sigma^\sharp \rangle \overset{\mathrm{b}}{\rightsquigarrow} m_0^\sharp \circledcirc m_1^\sharp}$$

$$\frac{\langle a_0, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_0^\sharp \quad \langle a_1, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m_1^\sharp}{\langle a_0 < a_1, \sigma^\sharp \rangle \overset{\mathrm{b}}{\rightsquigarrow} m_0^\sharp \olessthan m_1^\sharp} \qquad \frac{}{\langle \mathbf{skip}, \sigma^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma^\sharp}$$

$$\frac{\langle a, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m^\sharp}{\langle x := a, \sigma^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma^\sharp[x := a]} \text{(i)} \qquad \frac{\langle a, \sigma^\sharp \rangle \overset{\mathrm{a}}{\rightsquigarrow} m^\sharp}{\langle x := a, \sigma^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma^\sharp[m^\sharp/x]} \text{(ii)}$$

$$\frac{\langle s_0, \sigma_0^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_1^\sharp \quad \langle s_1, \sigma_1^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_2^\sharp}{\langle s_0; s_1, \sigma_0^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_2^\sharp}$$

$$\frac{\langle b, \sigma^\sharp \rangle \overset{\mathrm{b}}{\rightsquigarrow} t^\sharp \quad \bigl\langle s_0, \phi_{\mathrm{tt}}(\sigma^\sharp, b) \bigr\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_0^\sharp \quad \bigl\langle s_1, \phi_{\mathrm{ff}}(\sigma^\sharp, b) \bigr\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_1^\sharp}{\langle \mathbf{if}\, b\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_0^\sharp \sqcup \sigma_1^\sharp}$$

$$\frac{\langle b, \sigma^\sharp \rangle \overset{\mathrm{b}}{\rightsquigarrow} t^\sharp \quad \bigl\langle c, \phi_{\mathrm{tt}}(\sigma^\sharp, b) \bigr\rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_1^\sharp \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma_1^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \sigma_2^\sharp}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma^\sharp \rangle \overset{\mathrm{s}}{\rightsquigarrow} \phi_{\mathrm{ff}}(\sigma^\sharp, b) \sqcup \sigma_2^\sharp}$$

Notes:

(i) This rule is used if the domain $\mathrm{Store}^\sharp$ can capture the assignment precisely (e.g., when $\mathrm{Store}^\sharp$ is a domain of convex polyhedra and a is an affine expression). Notice that the premise is intentionally not used: its presence is required in order to ensure that the abstract tree approximates the concrete tree in its entirety.

(ii) This rule is used when (i) is not applicable.

Figure 4: Abstract semantics rule schemata for the simple imperative language

expansion of node m.[3]

3. Otherwise, there must be an ancestor node $m = \langle p, \sigma_m^\sharp \rangle \rightsquigarrow r_m$ of n labeled by the same syntax p, but the subsumption condition $\sigma_n^\sharp \sqsubseteq \sigma_m^\sharp$ does not hold. Then there are two options:
   (a) if the abstract domain $\mathrm{Store}^\sharp$ is finite, we proceed as in case (1);
   (b) if the abstract domain $\mathrm{Store}^\sharp$ is infinite, to ensure convergence, a widening '∇' over $\mathrm{Store}^\sharp$ can be employed and store $\sigma_n^\sharp$ in node n is replaced by $\sigma_m^\sharp \mathbin{\nabla} (\sigma_m^\sharp \sqcup \sigma_n^\sharp)$. Then, we proceed again as in case (1).

The abstract semantics of Figure 4 and the given algorithm for computing a rational abstract tree are fully generic in that any choice for the abstract domains $\mathrm{Int}^\sharp$, $\mathrm{Bool}^\sharp$ and $\mathrm{Store}^\sharp$ will result into a provably correct analysis algorithm. Focusing on numerical domains, the role of $\mathrm{Int}^\sharp$ can be played by any domain of intervals, so that the operations '$\oplus$', '$\ominus$' and '$\circledast$' are the standard ones of interval arithmetic [1]; for instance, $[m_0^{\mathrm{l}}, m_0^{\mathrm{u}}] \oplus [m_1^{\mathrm{l}}, m_1^{\mathrm{u}}] \stackrel{\mathrm{def}}{=} [m_0^{\mathrm{l}} + m_1^{\mathrm{l}}, m_0^{\mathrm{u}} + m_1^{\mathrm{u}}]$. More sophisticated domains, such as *modulo intervals* [64], are able to encode more precise information about the set of *integer* values each variable can take. For $\mathrm{Store}^\sharp$, a common choice is to abstract from the integrality of variables and consider a domain of convex polyhedra which, in exchange, allows the tracking of relational information. With reference to Figure 4, rule (i) can be applied directly when the arithmetic expression $a = \langle \mathbf{a}, \mathbf{x} \rangle + b$ is affine; the corresponding polyhedral operation is the computation of the image of a polyhedron by a special case of affine relation $\psi \subseteq \mathbb{R}^n \times \mathbb{R}^n$, called *single-update affine function*:

$$(\mathbf{v}, \mathbf{w}) \in \psi \iff w_k = \langle \mathbf{a}, \mathbf{v} \rangle + b \wedge \bigwedge_{\substack{0 \le i < n \\ i \ne k}} w_i = v_i.$$

Another special case, slightly more general than the one above and called *single-update bounded affine relation*, allows among other things to approximate non-linear assignments and to realize rule (ii). For fixed vectors $\mathbf{a}, \mathbf{c} \in \mathbb{R}^n$ and scalars $b, d \in \mathbb{R}$:

$$(\mathbf{v}, \mathbf{w}) \in \psi \iff \langle \mathbf{a}, \mathbf{v} \rangle + b \le w_k \le \langle \mathbf{c}, \mathbf{v} \rangle + d \wedge \bigwedge_{\substack{0 \le i < n \\ i \ne k}} w_i = v_i.$$

Both the rules for the if-then-else and the while constructs require the Boolean filters and least upper bound operations: these are realized by means of intersections (or the addition of individual constraints) and poly-hulls, respectively. These, together with the containment test used to detect the reaching of post-fixpoints and the widening (see Section 7) used to ensure termination of the analysis algorithm, are all the operations required for the analysis of our simple imperative language. More complex languages require other operations: for

[3]As explained in [70], the computation of such a *least* fixpoint (in the context of a coinductive interpretation of the abstract rules) is justified by the fact that here we only need to approximate the conclusions produced by the terminating concrete computations.

instance, the analysis of languages with command blocks needs to have the possibility of embedding polyhedra into a space of higher dimension, reorganizing the dimensions, and projecting polyhedra on spaces of lower dimension. Other operations are needed to accommodate different semantic constructions (e.g., affine preimages for backward semantics), to allow for the efficient modeling of data objects (e.g., summarized dimensions to approximate the values of unbounded collections [41]), and to help scalability (e.g., simplifications of polyhedra [38]).

Based on suitable variations of the simple linear invariant analysis outlined in this section (possibly combined with other analyses), many different applications have been proposed in the literature. Examples include the absence of common run-time arithmetic errors, such as floating-point exceptions, overflows and divisions by zero [21]; the absence of out-of-bounds array indexing [31, 78], as well as other buffer overruns caused by incorrect string manipulations [35, 37]; the analysis of programs manipulating (possibly unbounded) heap-allocated data structures, so as to prove the absence of several kinds of pointer errors (e.g., memory leaks) [41, 72]; the computation of input/output argument size relations in logic programs [18]; the detection of potential security vulnerabilities in x86 binaries that allow to bypass intrusion detection systems [57]; the inference of temporal schedulability constraints that a partially specified set of real-time tasks has to satisfy [34]. All of the above are examples of *safety* properties, whereby a computer program is proved to be free from some undesired behavior. However, the computation of invariant linear relations is also an important, often indispensable step when aiming at proving *progress* properties, such as termination [26, 61, 74]. It should be also stressed that the same approach, after some minor adaptations, can be applied to the analysis of alternative computation paradigms such as, e.g., *gated data dependence graphs* [53] (an intermediate representation for compilers) and *batch workflow networks* [77] (a form of Petri net used in workflow management).

4. Analysis and Verification of Hybrid Systems

Hybrid systems (that is, dynamical systems with both continuous and discrete components) are commonly modeled by hybrid automata [2, 38, 49]. These, often highly complex, systems are usually nonlinear (making them computationally intractable as they are). However, linear approximations, which allow the use of polyhedral computations for the model checking operations, have been used successfully for the verification of useful safety properties [36, 38, 75].

Definition4.1. (Linear hybrid automaton.) A *linear hybrid automaton* (of dimension n) is a tuple $(\mathrm{Loc}, \mathrm{Init}, \mathrm{Act}, \mathrm{Inv}, \mathrm{Lab}, \mathrm{Trans})$ where the first component Loc is a finite set of *locations.* The three functions $\mathrm{Init}\colon \mathrm{Loc} \to \mathbb{P}_n$, $\mathrm{Act}\colon \mathrm{Loc} \to \mathbb{P}_n$ and $\mathrm{Inv}\colon \mathrm{Loc} \to \mathbb{P}_n$ define polyhedra. In particular, for each location $\ell \in \mathrm{Loc}$: $\mathrm{Init}(\ell)$ specifies the set of possible initial values the n variables can take if the automaton starts at ℓ; $\mathrm{Act}(\ell)$ specifies the possible derivative values of the n variables, so that, if the automaton reaches ℓ with values given by the vector $\mathbf{v}$, then after staying there for a delay of $t \in \mathbb{R}_+$, the values will be given by a vector $\mathbf{v} + t\mathbf{w}$, where $\mathbf{w} \in \mathrm{Act}(\ell)$; $\mathrm{Inv}(\ell)$ specifies the values that an n-vector $\mathbf{v}$ may have at ℓ. The fifth and sixth components provide a set of *synchronization labels* Lab and a labeled set of affine transition relations $\mathrm{Trans} \subseteq \mathrm{Loc} \times \mathrm{Lab} \times \mathbb{P}_{2n} \times \mathrm{Loc}$, required to hold when moving from the *source* location (the first argument) to the *target* location (the fourth argument).

Observe that the only differences between this definition of a linear hybrid automaton and those in, for example [47, 49], are presentational; in particular, as we have used polyhedra to represent the linear constraints, there is no need to provide, as is the case in these other definitions, an explicit component of the system consisting of the set of n variables.

The synchronization labels Lab are required for specifying large systems. Each part of the system is specified by a separate automaton, and then *parallel composition* is employed to combine the components into an automaton for the complete system. This ensures that communication between the automata occurs, via selected input/output variables, between transitions that have the same label. Example 4.3 provides a very simple illustration of parallel composition; formal definitions are available in [2, 49].

A linear hybrid automaton can be represented by a directed graph whose nodes are the locations and edges are the transitions from the source to the target locations. Each node ℓ is labeled by two sets of constraints defining the polyhedra $\mathrm{Inv}(\ell)$ and $\mathrm{Act}(\ell)$. To distinguish these constraints, if, for example x is a variable used for the constraints defining $\mathrm{Inv}(\ell)$, $\dot{x}$ will be used in the constraints defining $\mathrm{Act}(\ell)$.[4] In the examples, the initial polyhedron $\mathrm{Init}(\ell)$ is assumed to be empty unless there is an arrow to ℓ (with no source node) labeled by the constraint system defining $\mathrm{Init}(\ell)$. Each edge $\tau = \bigl(\ell, a, \mathcal{P}, \ell'\bigr) \in \mathrm{Trans}$, is labeled by a constraint system $\mathcal{C}$ defining $\mathcal{P}$ and, optionally, by a which is only included where it is used for the parallel composition of automata. Since $\mathcal{P} \in \mathbb{P}_{2n}$, we specify $\mathcal{C}$ by using two n-tuples of variables $\mathbf{x}$ and $\mathbf{x}'$, which are interpreted as usual to denote the variables in the source ℓ and target ℓ' locations, respectively. We also adopt some helpful shorthand notation: $x{+}{+}$ and $x{-}{-}$ denote $x' = x + 1$ and $x' = x - 1$, respectively; also, constraints of the form $x' = x$ are omitted. The following examples, taken (with some minor modifications) from [2, 47], illustrate the automata.

Example4.2. A graphical view of a water-level monitor automaton is given in Figure 5. This models a system describing how the water level in a tank is controlled by a monitor that senses the water level w and operates a pump. When the pump is off, w falls by 2 cm per second; when the pump is on, w rises by 1 cm per second. However, there is a delay of 2 seconds from the moment the monitor signals the pump to change from on to off or vice versa before the switch is actually operated. Initially the automaton is at ℓ_0 with $w = 1$ and

[4]The dot notation reflects the fact that these variables denote the derivatives of the state variables.

it is required that $1 \leq w \leq 12$ at all times. Thus the monitor must signal the pump to turn on when $w = 5$ and signal it to turn off when $w = 10$.

The automaton illustrated in Figure 5 has 2 dimensions with variables w and x, where x denotes the time (in seconds) since the previous, most recent, signal from the monitor. There are four locations ℓ_i where $i = 0, 1, 2, 3$. At ℓ_0 and ℓ_1 the pump is on, while at ℓ_2 and ℓ_3 the pump is off. At ℓ_1 and ℓ_3 the monitor has signaled a change to the pump switch, but this has not yet been operated. Thus we have:

$$\begin{aligned}
\mathrm{Init}(\ell_0) &= \mathrm{con}\bigl(\{w = 1\}\bigr), \quad \mathrm{Init}(\ell_1) = \mathrm{Init}(\ell_2) = \mathrm{Init}(\ell_3) = \emptyset, \\
\mathrm{Inv}(\ell_0) &= \mathrm{con}\bigl(\{w < 10\}\bigr), \quad \mathrm{Inv}(\ell_1) = \mathrm{Inv}(\ell_3) = \mathrm{con}\bigl(\{x < 2\}\bigr), \\
\mathrm{Inv}(\ell_2) &= \mathrm{con}\bigl(\{w > 5\}\bigr), \quad \mathrm{Act}(\ell_0) = \mathrm{Act}(\ell_1) = \mathrm{con}\bigl(\{\dot{x} = \dot{w} = 1\}\bigr), \\
\mathrm{Act}(\ell_2) &= \mathrm{Act}(\ell_3) = \mathrm{con}\bigl(\{\dot{x} = 1, \dot{w} = -2\}\bigr).
\end{aligned}$$

There are four transitions $\tau_{ij} = (\ell_i, a_i, \mathcal{P}_i, \ell_j) \in \mathrm{Trans}$, where $i \in \{0, 1, 2, 3\}$ and $j = i + 1 \pmod 4$; the affine relations are

$$\begin{aligned}
\mathcal{P}_0 &= \mathrm{con}\bigl(\{w = w' = 10, x' = 0\}\bigr), & \mathcal{P}_1 &= \mathrm{con}\bigl(\{x = x' = 2, w' = w\}\bigr), \\
\mathcal{P}_2 &= \mathrm{con}\bigl(\{w = w' = 5, x' = 0\}\bigr), & \mathcal{P}_3 &= \mathcal{P}_1.
\end{aligned}$$

Example4.3. A representation of an automaton for a simple task scheduler is given in Figure 6. This models a scheduler with two classes of tasks A_1 and A_2, activated by interrupts I_1 and I_2. Interrupt I_1 (resp., I_2) occurs at most once every 10 (resp., 20) seconds and activates a task in class A_1 (resp., A_2), which takes 4 (resp., 8) seconds to complete. Tasks in A_2 have priority and preempt tasks in A_1. It is required that tasks in A_2 never wait.

The *Scheduler* automaton given in Figure 6 is the parallel composition of two component automata: *Interrupt* which models the assumptions about the interrupt frequencies; and *Task*, which models the execution of the tasks. The *Interrupt* automaton, which has a single location 'Intpt', has variables c_1 and c_2; c_i ($i = 1$, 2) measures the time elapsed since interrupt I_i occurred. The *Task* automaton has three locations: 'Idle' when no tasks are running; and 'Task1' and 'Task2' when tasks in classes A_1 (resp., A_2) are active. It has, for each $i = 1$, 2, variables x_i, which measures the execution time of task i, and k_i, which counts the number of pending tasks in class task i.

The combined *Scheduler* automaton has variables x_1, x_2, k_1, k_2, c_1 and c_2 and locations which are elements of the Cartesian product of the sets of locations for *Interrupt* and *Task*. As *Interrupt* has just one location, each *Task* location ℓ is used to denote the corresponding *Scheduler* location; here, the initial $\mathrm{Init}(\ell)$, derivative $\mathrm{Act}(\ell)$ and invariant $\mathrm{Inv}(\ell)$ polyhedra for the *Scheduler* are the concatenation of the corresponding component polyhedra for the *Task* and *Interrupt* automata (informally, a concatenation of polyhedra $\mathcal{P} \in \mathbb{P}_m$ and $\mathcal{Q} \in \mathbb{P}_n$ can be obtained by first embedding $\mathcal{P}$ into a vector space of dimension $n + m$ and then add a suitably renamed-apart version of the constraints defining $\mathcal{Q}$). Each transition $(\ell, a, \mathcal{P}, \ell')$ in the *Task* automaton not triggered

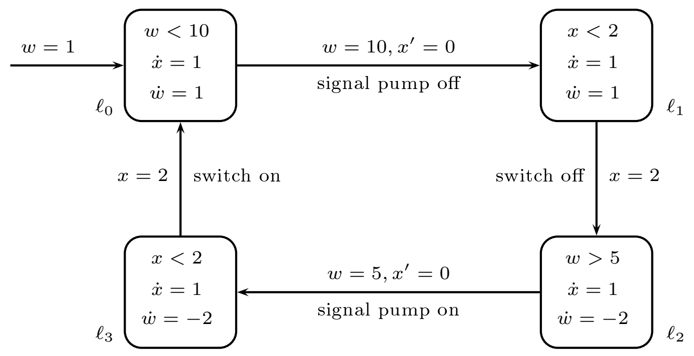


Figure 5: Water-level monitor

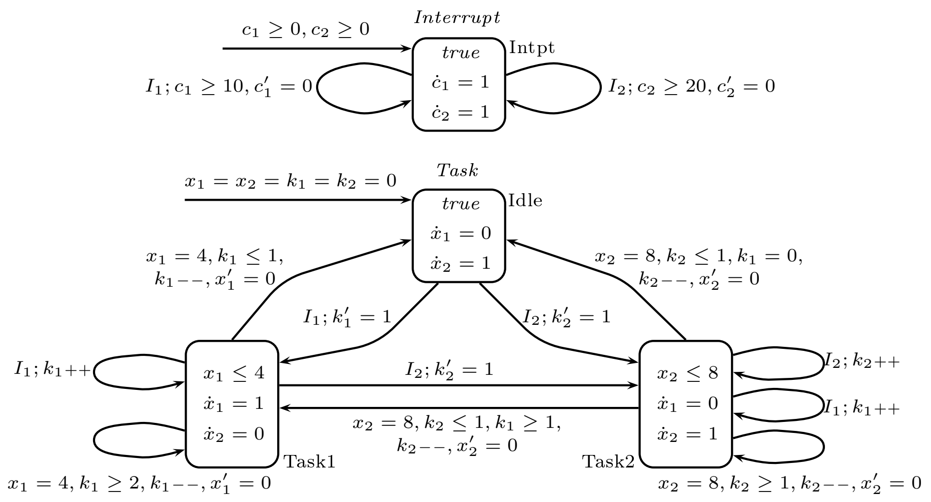


Figure 6: Scheduler

by interrupts I_1 and I_2 has a transition $(\ell, a, \mathcal{Q}, \ell')$ in the product automaton where $\mathcal{Q} \in \mathbb{P}_6$ is obtained by embedding $\mathcal{P}$ into a vector space of dimension 6. Letting $i = 1$, 2, for transitions $(\ell, I_i, \mathcal{P}, \ell')$ and $(\mathrm{Intpt}, I_i, \mathcal{P}', \mathrm{Intpt})$ in the *Task* and *Interrupt* automata, respectively, there is a transition $(\ell, I_i, \mathcal{Q}, \ell')$ in the product automaton where $\mathcal{Q} \in \mathbb{P}_6$ is obtained by concatenating $\mathcal{P}$ and $\mathcal{P}'$.

Given a linear hybrid automaton, the aim of an analyzer is to check, or even find sufficient conditions that ensure, that a valid *run* of the system cannot *reach* a location and vector of values that violate some requirement of the system. For instance, in Example 4.2, we need to show that the water level always lies between 1 cm and 12 cm; in Example 4.3, we need to show that no task in A_2 will ever wait. To show how polyhedral computations can be used to prove such properties, we first define more formally such a run and how reachable sets may be computed. Note that these definitions follow, with only minor changes, the approach in [47].

Letting $\mathcal{H} = (\mathrm{Loc}, \mathrm{Init}, \mathrm{Act}, \mathrm{Inv}, \mathrm{Lab}, \mathrm{Trans})$ be a linear hybrid automaton in n dimensions, a *state* s of $\mathcal{H}$ consists of a pair $(\ell, \mathbf{v})$, where $\ell \in \mathrm{Loc}$ and $\mathbf{v} \in \mathrm{Inv}(\ell)$. Given states $s = (\ell, \mathbf{v})$ and $s' = (\ell', \mathbf{v}')$, a time delay $t \in \mathbb{R}_+$ and a vector $\mathbf{w} \in \mathrm{Act}(\ell)$, $s \rightarrow^t_{\mathbf{w}} s'$ is a *step* of $\mathcal{H}$ provided that, for all $t' \in [0, t)$, $\mathbf{v} + t'\mathbf{w} \in \mathrm{Inv}(\ell)$ and, for some $(\ell, a, \mathcal{P}, \ell') \in \mathrm{Trans}$, $(\mathbf{v} + t\mathbf{w}) :: \mathbf{v}' \in \mathcal{P}$. A *run* of $\mathcal{H}$ is a sequence (finite or infinite) of steps $s_0 \rightarrow^{t_0}_{\mathbf{w}_0} s_1 \rightarrow^{t_1}_{\mathbf{w}_1} s_2 \cdots$, where the initial state $s_0 = (\ell_0, \mathbf{v}_0)$ satisfies the condition $\mathbf{v}_0 \in \mathrm{Init}(\ell_0)$. An infinite run diverges if the sum $\sum_{i \geq 0} t_i$ diverges. For each divergent run where, for $i \geq 0$, $s_i = (\ell_i, \mathbf{v}_i)$, we associate a (state) *behavior* β which is a total function from time to states: that is, $\beta(0) = s_0$ and, for each $t > 0$, $\beta(t) \stackrel{\mathrm{def}}{=} (\ell_i, \mathbf{v})$, where $i = \min\bigl\{ k \in \mathbb{N} \bigm| \sum_{j=0}^{k} t_j > t \bigr\}$ and $\mathbf{v} = \mathbf{v}_i + \mathbf{w}_i\bigl(t - \sum_{j<i} t_j\bigr)$. A state s is *reachable* if there exists a divergent run with behavior β and time $t \in \mathbb{R}_+$ such that $\beta(t) = s$. The set of all *reachable values* R_ℓ for a location ℓ is defined as:

$$R_\ell \stackrel{\mathrm{def}}{=} \bigl\{ \mathbf{v} \in \mathbb{R}^n \bigm| \exists t \in \mathbb{R}_+ \,.\, \beta(t) = (\ell, \mathbf{v}) \bigr\}.$$

The set of reachable values R_ℓ at a location ℓ can be characterized by a system of fixpoint equations that are defined in terms of sets of reachable values $R_{\ell'}$ at locations ℓ' where $\bigl(\ell', a, \mathcal{P}, \ell\bigr) \in \mathrm{Trans}$. These equations use the following operations on sets of vectors in $\mathbb{R}^n$. Let $\mathcal{P}, \mathcal{Q} \in \mathbb{P}_{2n}$ and $S \subseteq \mathbb{R}^n$. Then

$$\psi_{\mathcal{P}}(S) \stackrel{\mathrm{def}}{=} \{ \mathbf{v}' \in \mathbb{R}^n \mid \mathbf{v} \in S, \mathbf{v} :: \mathbf{v}' \in \mathcal{P} \};$$
$$S \nearrow \mathcal{Q} \stackrel{\mathrm{def}}{=} \{ \mathbf{v} + t\mathbf{w} \in \mathbb{R}^n \mid \mathbf{v} \in S, \mathbf{w} \in \mathcal{Q}, t \in \mathbb{R}_+ \}.$$

Note that, if $S \in \mathbb{P}_n$, then also $\psi_{\mathcal{P}}(S) \in \mathbb{P}_n$ and $S \nearrow \mathcal{Q} \in \mathbb{P}_n$. The '$\nearrow$' operator, called the *time elapse* operator, was first proposed in [47]. We can now provide the fixpoint equation for R_ℓ:

$$R_\ell = \Biggl(\biggl(\mathrm{Init}(\ell) \cup \bigcup_{(\ell', a, \mathcal{P}, \ell) \in \mathrm{Trans}} \psi_{\mathcal{P}}(R_{\ell'}) \cap \mathrm{Inv}(\ell)\biggr) \nearrow \mathrm{Act}(\ell)\Biggr) \cap \mathrm{Inv}(\ell). \quad (4.1)$$

Informally, the fixpoint equation for R_ℓ says that the reachable values at the location ℓ are obtained by letting the time elapse either from an initial value for ℓ or from a value obtained from an incoming transition. However, the fixpoint Equation (4.1) cannot handle strict constraints correctly and needs modifying; this is illustrated in the following example.

Example4.4. Consider again Example 4.2. Then, just applying Equation (4.1) (as proposed in [47]), the sets of reachable values at locations ℓ_1, ℓ_2, ℓ_3 are empty. The reason for this is that, for example, at location ℓ_0, the strict constraint $w < 10$ must hold, while in the transition from ℓ_0 to ℓ_1, the transition condition $w = 10$ has to hold. On the other hand, it follows from the definition of a step, that since one of the derivative constraints at ℓ_0 is $\dot{w} = 1$; the water level w may continue to increase up to the topological closure of R_{ℓ_0} which is consistent with $w = 10$.

To resolve this problem, in Equation (4.1) defining the concrete computation, $R_{\ell'}$ needs to be replaced by $c(R_{\ell'}) \cap \bigl(R_{\ell'} \nearrow \mathrm{Act}(\ell')\bigr)$, where $c(R'_\ell)$ denotes the topological closure of $R'_\ell \subseteq \mathbb{R}^n$.

Observe that, although the linear hybrid automata are specified by means of polyhedra, the reachable set R_ℓ for a linear hybrid automaton and location ℓ may not be a convex polyhedron since Equation (4.1) uses the set union operation. Therefore, to verify that some states of an automaton are unreachable using standard polyhedral computations, set union has to be replaced by the polyhull operation $\uplus$ described in Section 2. Thus the following fixpoint equation computes an approximation $R^\sharp_\ell$ to the reachability set R_ℓ.

$$R^\sharp_\ell = \biggl(\Bigl(\mathrm{Init}(\ell) \uplus \biguplus_{(\ell', a, \mathcal{P}, \ell) \in \mathrm{Trans}} \psi_{\mathcal{P}}(R^\sharp_{\ell'}) \cap \mathrm{Inv}(\ell)\Bigr) \nearrow \mathrm{Act}(\ell)\biggr) \cap \mathrm{Inv}(\ell). \quad (4.2)$$

As for the concrete fixpoint equation, to correctly handle the strict constraints in Equation (4.2) we need to replace $R^\sharp_{\ell'}$ with $c(R^\sharp_{\ell'}) \cap \bigl(R^\sharp_{\ell'} \nearrow \mathrm{Act}(\ell')\bigr)$.

If we let $\mathbf{R}^\sharp$ denote the tuple $\{\, R^\sharp_\ell \mid \ell \in \mathrm{Loc} \,\}$ we can write Equation (4.2) as $R^\sharp_\ell = F_\ell(\mathbf{R}^\sharp)$. For all $\ell \in \mathrm{Loc}$, we write $\mathbf{R}^{\sharp(0)}_\ell = \emptyset$ and, for all $k \geq 1$, $\mathbf{R}^{\sharp(k+1)}_\ell = F_\ell(\mathbf{R}^{\sharp(k)}_\ell)$. Then $\mathbf{R}^\sharp$ can be computed iteratively provided the sequence $\mathbf{R}^{\sharp(0)}, \mathbf{R}^{\sharp(1)}, \ldots$ does not diverge. To handle diverging sequences, we apply a widening (see Section 7.2); note that this only needs to be applied at sufficient locations so that each cyclic path in the graph of the hybrid automaton has at least one widening point.

Example4.5. Consider again Example 4.2. As there is a single loop passing through ℓ_0, it is sufficient to define the set of widening locations as $\{\ell_0\}$.

With the modified form of Equation (4.2) and the polyhedra widening of [31], the computation requires three iterations resulting in polyhedra defined by constraint systems $\mathcal{C}_i$ for $0 \leq i \leq 3$ where:

$$\mathcal{C}_0 = \{1 \leq w < 10\}, \qquad \mathcal{C}_1 = \{w - x = 10,\ 10 \leq w < 12\},$$
$$\mathcal{C}_2 = \{w + 2x = 16,\ 5 < w \leq 12\}, \qquad \mathcal{C}_3 = \{w + 2x = 5,\ 1 < w \leq 5\}.$$

Example4.6. Consider again Example 4.3. By applying the above mentioned polyhedra widening at location 'Task2' only, the analysis for the product automaton terminates in four iterations. After projecting onto variables k_1 and k_2, the reachable values are given by polyhedra defined by constraint systems $\mathcal{C}_{t0}$, $\mathcal{C}_{t1}$, and $\mathcal{C}_{t2}$ for locations 'Idle', 'Task1' and 'Task2', respectively, where:

$$\mathcal{C}_{t0} = \{k_1 = k_2 = 0\}, \quad \mathcal{C}_{t1} = \{k_2 = 0,\, k_1 = 1\}, \quad \mathcal{C}_{t2} = \{k_2 = 1\}.$$

Therefore, since at all locations $k_2 \leq 1$, no task in class A_2 will ever have to wait. However, as noted in [47], because of the convex hull approximation, with the polyhedral domain the analyzer fails to show that $k_1 \leq 2$. We therefore redid the analysis using a domain of powersets of polyhedra (see Section 6.2) and, after taking the poly-hull of the final sets and projecting onto variables k_1 and k_2, we obtained the polyhedra defined by constraint systems $\mathcal{C}'_{t0}$, $\mathcal{C}'_{t1}$ and $\mathcal{C}'_{t2}$ for locations 'Idle', 'Task1' and 'Task2', respectively, where:

$$\mathcal{C}'_{t0} = \{k_1 = k_2 = 0\}, \quad \mathcal{C}'_{t1} = \{k_2 = 0,\, k_1 = 1\}, \quad \mathcal{C}'_{t2} = \{k_1 \leq 2,\, k_2 = 1\}.$$

Hybrid systems with affine or nonlinear dynamics do not fit the above specification of a linear system so that the verification techniques described here are not directly applicable. Nonetheless, by partitioning the continuous state space and over-approximating the dynamics in each of the partitions, the same techniques used to verify linear hybrid automata can be used in these more general cases [36, 38, 51]. Such an approach has also been successfully applied in the verification of analog circuits, as discussed in the following section.

5. Analysis and Verification of Analog Systems

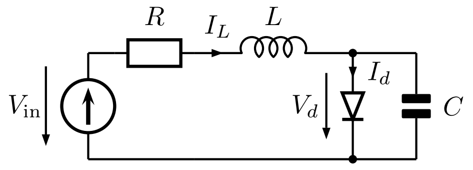
The idea of applying formal methods, that originated in the digital world, to analog systems was put forward in [48]. This is an important step forward with respect to more traditional methods for the validation of analog circuit designs. A formal verification tool can, for example, ensure that a design satisfies certain properties for entire sets of initial states and continuous ranges of circuit parameters, something that cannot be done with simulation.

To illustrate the approach, we describe a simple example of verification of an oscillator circuit, taken from [39]. To verify properties of the (cyclic) behavior of such circuits, cyclic invariants have to be determined. To establish a cyclic invariant for a given set of initial states and ranges for the circuit parameters, one has to show that the circuit returns to a subset of those initial states, which implies the system will keep traversing the same states indefinitely.

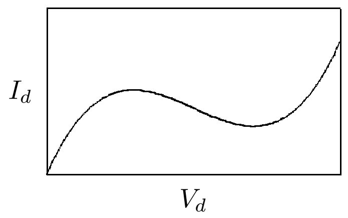
Consider the tunnel-diode oscillator schematized in Figure 7(a). The state of the system at a given instant of time is completely characterized by the values of the inductor current I_L and the diode voltage drop V_d. With these as the state variables, the system is described by the second-order state equations

$$\dot{V}_d = 1/C\bigl(-I_d(V_d) + I_L\bigr), \tag{5.1}$$

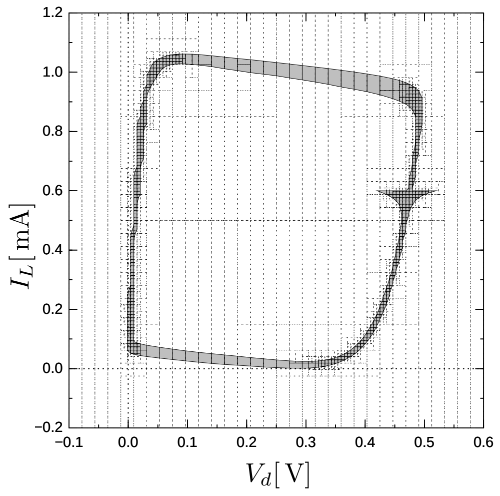
$$\dot{I}_L = 1/L(-V_d - RI_L + V_{\text{in}}). \tag{5.2}$$



(a) Circuit schematic



(b) Tunnel diode characteristic



(c) Reachable states (dashed)

(Picture courtesy of Goran Frehse.)

Figure 7: Tunnel-diode oscillator circuit

In [39] it is shown how a cyclic invariant can be obtained for this circuit using the PHAVer system. First, a piecewise affine envelope is constructed for the tunnel diode characteristic $I_d(V_d)$ depicted in Figure 7(b): for the particular example analyzed in [39], sufficient precision is obtained by subdividing the range $V_d \in [-0.1\,\mathrm{V}, 0.6\,\mathrm{V}]$ into 64 intervals, resulting in a piecewise affine model of (5.1). Forward reachability computation with PHAVer can obtain the set of states depicted in Figure 7(c). These are the states reachable from the set of initial states corresponding to $V_d \in [0.42\,\mathrm{V}, 0.52\,\mathrm{V}]$ and $I_L = 0.6\,\mathrm{mA}$ (the base of the downward-facing triangular shape in Figure 7(c)). As the loop shape constituted by the reachable states is traversed clockwise, it can be seen that the inductor current I_L returns to the initial value of $0.6\,\mathrm{mA}$ with a diode voltage drop that is well within the initial range $[0.42\,\mathrm{V}, 0.52\,\mathrm{V}]$. The set of reachable states so obtained is thus an invariant of the circuit.

In [39] it is shown that, due to over-approximation, forward reachability can fail to determine invariants of more complex circuits. A new technique combining forward and backward reachability with iterative refinement of the partitions is thus proposed and shown to be more powerful and efficient.

6. Families of Polyhedral Approximations

For several applications of static analysis and verification, an approximation based on the domain of convex polyhedra can be regarded as the most appropriate choice. In this section we discuss alternative options (simplifications, generalizations, and combinations with other numerical domains) that might be considered when trying either to reduce the cost of the analysis, or to increase the precision of the computed results.

6.1. Simplifications of Polyhedra

There are contexts where approximations based on general convex polyhedra, no matter which implementation is adopted, incur an unacceptable computational cost. In such cases, the static analysis may resort to further simplifications so as to obtain useful results within reasonable time and space bounds.

A first, almost traditional approach is based on the identification of suitable syntactic subclasses of polyhedra. The abstract domain of *bounding boxes* (or intervals [27]) is based on polyhedra that can be represented as finite conjunctions of constraints of the form $\pm x_i \leq d$ or $\pm x_i < d$, leading to the specification of operations whose worst-case complexity is linear in the number of space dimensions. As a more precise alternative, the class of *potential constraints* [16], also known as *bounded differences* [6, 32], allows for constraints of the form $x_i - x_j \leq d$ or $\pm x_i \leq d$; the abstract domain of *octagons* [62] also admits constraints of the form $x_i + x_j \leq d$. In these last two cases, the operators are characterized by a worst-case time complexity which is cubic in the number of space dimensions. For all of the approximations mentioned above, improved efficiency also follows from the fact that the corresponding computations are simple enough to allow for the adoption of floating-point data types: in contrast, the specification of safe and efficient floating-point operations for general polyhedra is an open problem, so that polyhedra libraries have to be based on unbounded precision data types.

Several alternative (syntactic and/or semantic) simplification schemes have been put forward in the recent literature. The *Two Variables per Linear Inequality* abstract domain is proposed in [73], where constraints take the syntactic form $ax_i + bx_j \leq d$. In [69], an arbitrary family of polyhedra is chosen before starting the analysis by fixing the slopes of a finite number of linear inequalities, which are called the *template constraints*; linear programming techniques are then used to compute precise approximations in the considered class of shapes. In contrast, in [68], general polyhedra are allowed, but the corresponding operations (in particular, the poly-hull and the image of affine relations) are approximated by less precise variants so as to ensure a polynomial worst-case complexity in the size of the inputs. An even more flexible approach is proposed in [38], where arbitrary polyhedra are approximated, when they become too complex, by limiting the number of constraints in their description and/or the magnitude of the coefficients occurring in the constraints. These more dynamic approximation schemes are promising, in particular for those applications where nothing is known in advance about the syntactic form of the constraints that will be computed during the analysis.

An important observation to be made is that there is no actual need to prefer *a priori* (and therefore commit to) a specific abstract domain: the analysis tool may be based on several abstractions, safely switching from more precise, possibly costly domains to more efficient, possibly imprecise ones, and vice versa, depending on the context. When replacing a generic polyhedron by a simpler one, the problem of the identification of a good over-approximation has to be solved. Depending on the context, the approaches may vary significantly. At one extreme, when efficiency is really critical, the adoption of syntactic techniques should be pursued: for an interesting example, we refer the reader to one of the simplification heuristics used in [38], where the efficient selection of a small number of linear inequalities out of a constraint system is driven by a simple, yet effective reasoning on the measure of the angles formed by the corresponding half-spaces. At the other extreme, linear programming (LP) optimization techniques may be used so as to obtain the best match in the considered class of geometric shapes. For instance, the precise approximation of a polyhedron by a bounding box (resp., a bounded difference or octagon) can be implemented by a linear (resp., quadratic) number of optimizations of a class of LP problems, where the objective function varies while the feasible region is invariant and defined by the constraints of the polyhedron. Note that, if correctness has to be preserved, it is essential that no rounding error is made on the wrong side, so that classical floating-point implementations of LP solvers have to be considered unsafe, unless the computed results can be certified by some other tool. Alternatively, it is possible to consider LP implementations based on unbounded precision data types.

When the number of space dimensions to be modeled is beyond a given threshold, the whole analysis space can be split into a finite number of smaller, more manageable components, thereby realizing a further simplification scheme that can be combined with those described above. The splitting strategy varies considerably. In [46], Cartesian factoring techniques are used so as to dynamically partition the space dimensions of a polyhedron into independent subsets; the orthogonal factors are then approximated by lower dimensional polyhedra with no precision penalty. In an alternative approach described in [21], many (possibly overlapping) small subsets of space dimensions, called *variable packs*, are identified before the start of the analysis by means of syntactic conditions; the relations holding between the variables in each pack are then approximated by using an octagonal abstraction. A variation of this is described in [78], where non-overlapping variable packs are dynamically computed (and possibly merged) during the analysis, whereas the relations between the variables in a pack are approximated by means of potential constraints. In [78] it is also observed that, since the average size of variables packs is small (5 variables), more precise approximations based on general polyhedra should be feasible.

6.2. Generalizations of Polyhedra

There are applications where the restriction to the domain of convex polyhedra is intrinsically inadequate. This may happen, not only when the verification property of interest is itself non-convex, but also when the adopted computation strategy requires that a convex property is proved by passing through a non-convex intermediate approximation. This was the case in Example 4.6 of Section 4, where the upper bound ($k_1 \leq 2$) on the number of waiting processes for class A_1 was obtained by switching from the domain of convex polyhedra to the domain of finite sets of polyhedra.

The finite powerset domain construction [7] is a special case of *disjunctive completion* [28], a systematic technique to derive an enhanced abstract domain starting from an existing one. A finite powerset domain implements disjunctions by maintaining an explicit (hence finite) and *non-redundant* collection of elements of the base-level domain: non-redundancy means that a collection is made of maximal elements with respect to the approximation ordering, so that no element subsumes another element in the collection.

For a better understanding of the concepts, which are described in completely general terms in [12], let us consider the application of the finite powerset construction to the domain of convex polyhedra. This instantiation (which is the one also adopted for the examples developed in [12]) can be used to model nonlinear systems as described, e.g., in Section 5. Then, an element of the abstract domain is a finite set of maximal convex polyhedra, so that no polyhedron in the set is contained in another polyhedron in the set. The powerset domain is a lattice: the bottom and top elements are $\emptyset$ and $\{\mathbb{R}^n\}$, respectively; the meet is obtained by removing redundancies from the set of all possible binary intersections of an element in the first powerset with an element in the second powerset; while the binary join is the non-redundant subset of the union of the two arguments. Most of the other abstract operations needed for a static analysis using the finite powerset domain are easily obtained by "lifting" the corresponding operations defined on the base-level domain, and then reinforcing non-redundancy. For instance, the computation of the image of a finite powerset under an affine relation is obtained by computing the image of each polyhedron in the collection. However, the construction of a provably correct widening operator has only recently been addressed in [12] (see Section 7.2). The generic specification of the abstract operators of the finite powerset domain in terms of abstract operations on the (arbitrary) base-level domain allows for the development of a single implementation which is shared by all the possible instances of the domain construction.

An alternative abstraction scheme has been proposed in [15] for the case of finite conjunctions of polynomial inequalities. Intuitively, a polynomial constraint can be approximated by means of a linear constraint in a higher dimension vector space, so that the different terms of the polynomial (e.g., x_0, x_0x_1, x_0^2) are mapped to different and independent space dimensions; these linear constraints are then used to perform an almost classical linear relation analysis based on convex polyhedra. Due to the linearization step, most of the precision of the polynomial constraints is initially lost; however, some of the relations holding between the different terms of the original polynomial can be recovered by adding further constraints that are redundant when interpreted in the polynomial world, but do contribute to precision in the linearized space. In particular, in [15] the polynomial constraints are mapped into finitely generated *polynomial cones* and a degree-bounded product closure operator is systematically applied so as to improve accuracy. As a trivial example, let the polynomial terms x_0, x_1 and x_0x_1 be mapped to the space dimensions y_0, y_1 and y_2, respectively. Then, the linearization of the polynomial constraints $x_0 \geq 0$ and $x_1 \geq 0$ will produce a polyhedron that, while satisfying $y_0 \geq 0$ and $y_1 \geq 0$, leaves variable y_2 totally unconstrained. By applying the product closure operator we also obtain the linear constraint $y_2 \geq 0$, thereby recovering the non-negativity of term $x_0 x_1$.

6.3. Combinations with other Numerical Abstractions

There are two basic kinds of numerical abstractions for approximating the values of the program variables: outer *limits* (or bounds within which the values must lie) and the pattern of *distribution* of these values. The first can be approximated by (constructions based on) convex polyhedra, while the second can be approximated by sets of congruences defining lattices of points we call *grids* [8, 43]. Before considering how these and similar domains may be combined, we give a brief overview of the domain of grids.

Any vector that satisfies $\langle \mathbf{a}, \mathbf{v} \rangle = b + \mu f$, for some $\mu \in \mathbb{Z}$, is said to *satisfy* the congruence relation $\langle \mathbf{a}, \mathbf{v} \rangle \equiv_f b$. A *congruence system* $\mathcal{K}$ is a finite set of congruence relations in $\mathbb{R}^n$. A *grid* is the set of all vectors in $\mathbb{R}^n$ that satisfy the congruences in $\mathcal{K}$. The domain of grids $\mathbb{G}_n$ is the set of all grids in $\mathbb{R}^n$ ordered by the set inclusion relation, so that the empty set and $\mathbb{R}^n$ are the bottom and top elements of $\mathbb{G}_n$ respectively and the intersection of two grids is itself a grid. Thus, as for the domain of polyhedra, the domain of grids forms a lattice $(\mathbb{G}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap)$ where $\uplus$ denotes the join operation returning the least grid greater than or equal to the two arguments. For more details concerning all aspects of the domain of grids, see [8].

The distribution information captured by grids has a number of applications in its own right: for instance, to ensure that external memory accesses obey the alignment restriction imposed by the host architecture, and to enable several transformations for efficient parallel execution as well as optimizations that enhance cache behavior. However, here we are primarily concerned with applications that can benefit from the combination of the domain of grids with that of convex polyhedra. For instance, knowing the frequency (and position) of the points in a grid, we can shrink the polyhedra so that the bounding hyperplanes pass through the grid values; if this leads to a polyhedron with reduced dimension (such as a single point) or one that is empty, it can lead, not only to improved precision, but also a more efficient use of resources by the analyzer [3, 65, 67].

Generic constructions, such as direct and reduced product, can be used to provide a formal basis for the combination of the grid and polyhedral domains [28] although the exact choice of product construction used to build the grid-polyhedral domain needs further study. Both the direct and reduced products have problems: the direct product has no provision for communication between the component domains, thereby losing precision; while the reduced product, which is the most precise refinement of the direct product, has exponential complexity. It is expected that, for grid-polyhedra, the most useful product construction will lie between these extremes. For instance, as equalities are common entities for both constraint and congruence systems, if an equality is found to hold in one component, it is safe to just add this to the other component. In addition, in an element of the grid-polyhedral domain, any hyperplane that bounds the polyhedron component could be moved inwards until it intersects with points of the grid with only linear cost on the number of dimensions. Of course, this reduction on its own is not optimal since the grid points in the intersection may not lie in the polyhedron itself. For optimality or, more generally, so as to gain additional precision, we need to experiment with various forms of the branch-and-bound and cutting-plane algorithms [56] already well-researched for integer linear programming. What is needed is a range of options for the product construction allowing the user to decide on the complexity/precision trade-off. Further work on this is needed, including an investigation of other proposals for generic products that lie between the direct and reduced product, such as the local decreasing iteration method [42] and the open product construction [25].

7. Polyhedral Computations Peculiar to Analysis and Verification

As observed in the previous sections, the analysis of the run-time behavior of a system can be broken down into a set of basic operations on the chosen abstract domains. This means that each abstract domain should provide adequate computational support for such a set and, where appropriate, further operations that might be useful for tuning the cost/precision ratio. In this section, we discuss several key issues relevant to the design and implementation of an abstract domain of, or based on, convex polyhedra. Before going into further detail, it should be stressed that the particular context of the application plays a significant and non-trivial role here. For instance, in many computational complexity studies, it is assumed that a small number of operations (often, just a single one) can have arbitrarily large operands; also, it is typically required that exact results have to be computed. These assumptions taken together may be inappropriate in the context of static analysis: it is quite often the case that a large number of operations will have only small or medium sized operands; also, whenever facing an efficiency issue, the exactness requirement can be dropped (provided soundness is maintained). As a consequence, the evaluation of alternative algorithmic strategies should be largely based on practical experimentation.

7.1. The Double Description Method

Convex polyhedra are typically specified by a finite system of linear inequality constraints and for this representation there are known algorithms (e.g., based on Fourier-Motzkin elimination [58, 71]) for most of the operations already mentioned.

An alternative approach is based on the *double description* method due to Motzkin et al. [63]. This method was originally defined on the set of topologically closed convex polyhedra, a sub-lattice $(\mathbb{CP}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cup)$ of the lattice of (not necessarily closed, or NNC) polyhedra $\mathbb{P}_n$. In the double description method, a closed polyhedron may be described by using a system of non-strict linear inequalities or by using a *generator* system that records its key geometric features. The following is the main theoretical result, which is a simple consequence of well-known theorems by Minkowski and Weyl [76].

Theorem 7.1. *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a topologically closed convex polyhedron if and only if there exist finite sets $R, P \subseteq \mathbb{R}^n$ of cardinality r and p, respectively, such that $\mathbf{0} \notin R$ and $\mathcal{P}$ can be generated from (R, P) as follows:*

$$\mathcal{P} = \{\, R\rho + P\pi \in \mathbb{R}^n \mid \rho \in \mathbb{R}_+^r, \pi \in \mathbb{R}_+^p, \textstyle\sum_{i=1}^p \pi_i = 1 \,\}.$$

Intuitively, a point of a polyhedron $\mathcal{P}$ is obtained by adding a convex combination of the vectors in P (the generating points) to a conic combination of the vectors in R (the generating *rays*).

It turns out that constraint and generator descriptions are duals: each representation can be computed starting from the other one. Clever implementations of this conversion procedure, improving on the Chernikova's algorithm [23], are the starting point for the development of software libraries that, while being characterized by a worst case computational cost which is exponential in the size of the input, turn out to be practically useful. A common characteristic of these implementations is the exploitation of *incrementality*, whereby most of the computational work done for an operation is reused to efficiently compute small variations of the corresponding result. Further computational enhancements are obtained by the adoption of suitable heuristics, ranging from the efficient handling of adjacency information [59], to a careful choice of ordering strategies for the computation of intermediate results [4, 5, 40]; the overall construction typically relies on a tight integration of the basic algorithms with a carefully chosen set of data structures [14].

An important motivation for the adoption of an implementation based on the double description method is that the ability to switch from a constraint description to a generator description, or vice versa, can be usefully exploited to provide simple implementations for the basic operations on polyhedra. For instance, set intersection is easily implemented by taking the union of the constraint systems representing the two arguments, whereas the poly-hull is implemented by joining the generator systems representing the two arguments; and the test for emptiness can be implemented by checking that the generator system has no points. Moreover, a test for subset inclusion $\mathcal{P} \subseteq \mathcal{Q}$ can be implemented by checking if each point and each ray in a generator system describing $\mathcal{P}$ satisfies all linear inequalities in a constraint system describing $\mathcal{Q}$. As a further example, the time elapse operation specified in Section 4, can be implemented using the generator systems for the argument polyhedra [47]. That is a generator system for the polyhedron $\mathcal{P} \nearrow \mathcal{Q}$ can be obtained by adopting the same set of generating points as $\mathcal{P}$ and by defining its set of rays as the union of the set of generating rays for $\mathcal{P}$ with the set of all the generators (both points and rays) for $\mathcal{Q}$.

As seen in Section 3, in the context of the analysis of imperative languages one of the most frequent statements is variable assignment, where the expression assigned is safely approximated by an affine relation $\psi \subseteq \mathbb{R}^n \times \mathbb{R}^n$. The (direct or inverse) image of an affine relation can be naively computed by embedding the input polyhedron $\mathcal{P} \subseteq \mathbb{R}^n$ into the space $\mathbb{R}^{2n}$, intersecting it with the constraints defining ψ and finally projecting the result back on $\mathbb{R}^n$. However, due to the moves to/from a higher dimensional space, this approach suffers from significant overheads. Quite often, the expression assigned is a simple affine function of the variables' values and can thus be exactly modeled by computing the image of a single-update affine function. With the double description method, the images of affine functions are much more efficiently computed by applying them directly to the generators of the argument polyhedron. A dual approach, using the constraint description of the polyhedron, allows for the computation of the preimages of affine functions, which can be of interest for a backward semantic construction, where the initial values of program variables are approximated starting from their final values. Similar efficiency arguments motivate the study of specific implementations for single-update bounded affine relations and other special subclasses of affine relations.

7.2. Widening and Narrowing

The first widening operator for the domain of convex polyhedra, the so-called *standard widening* proposed in [31], can be informally described as follows: suppose that in the post-fixpoint iteration sequence we compute as successive iterates the polyhedra $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$; then, the widening keeps all and only the constraints defining $\mathcal{P}_i$ that are also satisfied by $\mathcal{P}_{i+1}$. This simple idea, which is basically borrowed from the widening operator defined on the domain of intervals [27], is quite effective in ensuring the termination of the analysis (the number of constraints decreases at each iteration); by avoiding the application of the widening in the first few iterations of the analysis and/or by applying the "widening up-to" technique of [45], it also provides, in the main, an adequate level of precision.

Some application fields, however, are particularly sensitive to the precision of the deduced numerical information, to the point that some authors propose to give up the termination guarantee and use so-called *extrapolation* operators: examples include the operators defined in [50] and [52], as well as the proposals in [22] and [33] for sets of polyhedra and the heuristics sketched in [19].

In [10] this precision problem is reconsidered in a more general context and a framework is proposed that is able to improve upon the precision of a given widening while keeping the termination guarantee. The approach, which builds on theoretical results put forward in work on termination analysis, combines an existing widening operator, whose termination guarantee should be *formally certifiable*, with an arbitrary number of precision improving heuristics. Its feasibility was demonstrated by instantiating the framework so as to produce a new widening on polyhedra improving upon the precision of the standard widening in a significant percentage of benchmarks.

For the more challenging case of an abstract domain obtained by the finite powerset domain construction, several generic schemes of widenings have been proposed in [12] that are able to "lift" a widening defined on the base-level domain without compromising its termination guarantee. The instantiation of such a generic approach led to the definition of the first non-trivial and provably correct widenings on a domain of finite sets of convex polyhedra. Being highly parametric, the widening schemes proposed in [12] can be instantiated according to the needs of the specific application, as done in [44]. One of the heuristic approaches adopted in [12] to control the precision/complexity trade-off of the widenings, originally proposed in [22], attempts at reducing the cardinality of a polyhedral collection by merging two of its elements whenever their set union happens to be a convex polyhedron. The implementation of such a heuristic could significantly benefit from the results and algorithms presented in [17].

It is also worth mentioning that, once a post-fixpoint approximation has been obtained by means of an upward iteration sequence with widening, its precision can be improved by means of a downward iteration, possibly using a *narrowing operator* [27, 29]. To the best of our knowledge, no narrowing has ever been defined on the domain of convex polyhedra: applications simply stop the downward computation after a small number of iterations.

7.3. Not Necessarily Closed Convex Polyhedra

Most static analysis applications computing linear inequality relations between program variables consider the domain $\mathbb{CP}_n$ of topologically closed polyhedra. One of the underlying motivations is that sometimes (e.g., when working with integer valued variables only) strict inequalities can be filtered away by suitable syntactic manipulations; even when this is not the case, the topological closure approximation may be interpreted as a quick and practical workaround to the fact that some software libraries do not fully support computations on NNC polyhedra. However, there are applications [2, 24, 47] where the ability of encoding and propagating strict inequalities might be crucial for the usefulness of the final results.

The first proposal for a systematic implementation of strict inequalities in a software library based on the double description method was put forward in [47]: a syntactic translation embeds an n-dimensional NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ into an $(n+1)$-dimensional closed polyhedron $\mathcal{R} \in \mathbb{CP}_{n+1}$, by adding a single *slack variable* ϵ, satisfying the additional side constraints $0 \leq \epsilon \leq 1$. Namely, any strict inequality constraint $\langle \mathbf{a}, \mathbf{x} \rangle > b$ is translated into the non-strict inequality constraint $\langle \mathbf{a}, \mathbf{x} \rangle - \epsilon \geq b$. The computation is thus performed on the closed representation $\mathcal{R} \in \mathbb{CP}_{n+1}$, with only minor adaptations to the basic algorithms so as to take into account the *implicit* strict constraint $\epsilon > 0$.

While this idea is quite effective, the resulting software library no longer enjoys all of the properties of the underlying double description implementation: NNC polyhedra cannot be suitably described using generator systems, and the geometric intuitions are lost under the "implementation details." These problems motivated the studies in [11], where a proper generalization of the double description method to NNC polyhedra was proposed. The main improvement was the identification of the *closure point* as a new kind of generator for NNC polyhedra, leading to the following result generalizing Theorem 7.1:

Theorem 7.2. *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is an NNC polyhedron if and only if there exist finite sets $R, P, C \subseteq \mathbb{R}^n$ of cardinality r, p and c such that $\mathbf{0} \notin R$ and*

$$\mathcal{P} = \left\{ R\rho + P\pi + C\gamma \;\middle|\; \rho \in \mathbb{R}^r_+, \pi \in \mathbb{R}^p_+ \setminus \{\mathbf{0}\}, \gamma \in \mathbb{R}^c_+, \textstyle\sum_{i=1}^{p} \pi_i + \sum_{i=1}^{c} \gamma_i = 1 \right\}.$$

The new condition $\pi \neq \mathbf{0}$ ensures that at least one of the points of P plays an active role in any convex combination of the vectors of P and C. As a consequence, the vectors of C are closure points of $\mathcal{P}$, i.e., points that belong to the topological closure of $\mathcal{P}$, but may not belong to $\mathcal{P}$ itself.

Thanks to the introduction of (strict inequalities and) closure points, most of the pros of the double description method now also apply to the domain of NNC polyhedra [11]: simpler, higher-level implementations of operations on NNC polyhedra can be specified, reasoned about and justified in terms of any one of the two dual descriptions; important implementation issues (such as the need to identify and remove all kinds of redundancies in the descriptions) can be provided with proper solutions; different lower-level encodings (e.g., an alternative management of the slack variable) can be investigated and experimented with, without affecting the user of the software library. It would be interesting, from both a theoretical and practical point of view, to provide a more direct encoding of NNC polyhedra, i.e., one that is not based on the use of slack variables; this requires the specification and the corresponding proof of correctness of a direct NNC conversion algorithm, potentially achieving a major efficiency improvement.

8. Conclusion

In the field of automatic analysis and verification of software and hardware systems, approximate reasoning on numerical quantities is crucial. As first recognized in 1978 [31], polyhedral computation algorithms can be used for the automatic inference of numerical assertions that correctly (though usually not completely) characterize the behavior of a system at some level of abstraction.

Until the end of the 1990's these techniques were not in widespread use, mainly due to the unavailability of robust and efficient implementations of convex polyhedra. As far as we know, the first published libraries of polyhedral algorithms suitable for analysis and verification purposes have been *Polylib*, released in 1995, written by Wilde at IRISA [79] and based on earlier work by Le Verge [59], and the polyhedra library of *POLINE* (POLyhedra INtegrated Environment) written by Halbwachs and Proy at Verimag and also released in 1995. Both libraries used machine integers to represent the coefficients of linear equalities and inequalities, something that could easily result into (undetected) overflows. While Polylib provided only a fraction of the functionalities offered by POLINE's library (which offered, among other things, support for NNC polyhedra), it was available in source format. The POLINE's library, instead, was distributed only in binary form for the Sun-4 platform (freely, until about the year 1996; under rather restrictive conditions afterward). POLINE included also a system called POLKA (POLyhedra desK cAlculator) and an analyzer for linear hybrid automata. A variation of a subset of POLINE's library was incorporated into the *HyTech* tool [51].

The work of Wilde and Le Verge, which was extended by Loechner [60], led to the creation of *PolyLib*. The *New Polka* library by Jeannet, first released in 2000 and originally based on both IRISA's Polylib and POLINE's library, incorporates the idea —suggested by Fukuda and Prodon [40]— of lexicographically sorting the matrices representing constraints and generators. New Polka, which supports both closed and NNC polyhedra, together with Miné's *Octagon Abstract Domain Library* [62] and an interval library called *ITV*, is now included in the *APRON* library. Finally, the *Parma Polyhedra Library* (PPL), initially inspired by New Polka and first released in 2001, is developed and maintained by the authors of this paper. The PPL supports both closed and NNC polyhedra, bounding boxes, bounded difference and octagonal shapes, grids and combinations of the above including the finite powerset construction [14].

The above libraries have all been designed specifically for applications of analysis and verification such as those described in this paper. However, two libraries that were designed for solving vertex enumeration/convex hull problems have successfully been used in static analysis and computer-aided verification tools: Fukuda's *cddlib*, an implementation of the double description method [63]; and *lrslib*, the implementation by Avis of the reverse search algorithm [4].

All the libraries mentioned in the last two paragraphs are distributed under free software licenses and support the use of unbounded numeric coefficients. This, together with the ever increasing available computing power and the growing interest in ensuring the correctness of critical systems, has caused, in the 2000's, the continuous emergence of new tools and applications of polyhedral computations in the area of formal methods. As a consequence, this is much more of a new beginning than an end to research in this area. As explained in Sections 6 and 7, several open issues remain. Most of them have to do with the need for effectively managing the complexity-precision trade-off: the encouraging results obtained with today's tools are pushing us to apply them to more complex systems for a possibly more precise analysis and/or verification of more complex properties.

Acknowledgments We thank Goran Frehse for the discussion we had on polyhedra simplifications and for the PostScript code for Figure 7(c).

References

[1] G. Alefeld, J. Herzberger, Introduction to Interval Computation, Academic Press, New York, 1983.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, P.-H. Ho, Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems, in: Hybrid Systems I, vol. 736 of Lecture Notes in Computer Science, 1993.

[3] C. Ancourt, Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales, Ph.D. thesis, Université de Paris VI, Paris, France (Mar. 1991).

[4] D. Avis, lrs: A revised implementation of the reverse search vertex enumeration algorithm, in: G. Kalai, G. M. Ziegler (eds.), Polytopes — Combinatorics and Computation, vol. 29 of Oberwolfach Seminars, Birkhäuser-Verlag, 2000, pp. 177–198.

[5] D. Avis, D. Bremner, How good are convex hull algorithms?, in: Proceedings of the Eleventh Annual Symposium on Computational Geometry, ACM Press, Vancouver, B.C., Canada, 1995.

[6] R. Bagnara, Data-flow analysis for constraint logic-based languages, Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, printed as Report TD-1/97 (Mar. 1997).

[7] R. Bagnara, A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages, Science of Computer Programming 30 (1–2) (1998) 119–155.

[8] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, E. Zaffanella, Grids: A domain for analyzing the distribution of numerical values, in: G. Puebla (ed.), Logic-based Program Synthesis and Transformation, 16th International Symposium, vol. 4407 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Venice, Italy, 2007.

[9] R. Bagnara, P. M. Hill, A. Pescetti, E. Zaffanella, On the design of generic static analyzers for modern imperative languages, Tech. Rep. `arXiv:cs.PL/0703116`, Dipartimento di Matematica, Università di Parma, Italy, available from `http://arxiv.org/` (2007).

[10] R. Bagnara, P. M. Hill, E. Ricci, E. Zaffanella, Precise widening operators for convex polyhedra, Science of Computer Programming 58 (1–2) (2005) 28–56.

[11] R. Bagnara, P. M. Hill, E. Zaffanella, Not necessarily closed convex polyhedra and the double description method, Formal Aspects of Computing 17 (2) (2005) 222–257.

[12] R. Bagnara, P. M. Hill, E. Zaffanella, Widening operators for powerset domains, Software Tools for Technology Transfer 8 (4/5) (2006) 449–466. (As the figures in the journal version of this paper have been improperly printed —rendering them useless—, we recommend that interested readers download an electronic copy from the PPL's web site at `http://www.cs.unipr.it/ppl/`.)

[13] R. Bagnara, P. M. Hill, E. Zaffanella, Applications of polyhedral computations to the analysis and verification of hardware and software systems, `arXiv:cs.CG/0701122`, available from `http://arxiv.org/`. (2007).

[14] R. Bagnara, P. M. Hill, E. Zaffanella, The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems, Science of Computer Programming 72 (1–2) (2008) 3–21.

[15] R. Bagnara, E. Rodríguez-Carbonell, E. Zaffanella, Generation of basic semi-algebraic invariants using convex polyhedra, in: C. Hankin, I. Siveroni

(eds.), Static Analysis: Proceedings of the 12th International Symposium, vol. 3672 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, London, UK, 2005.

[16] R. Bellman, Dynamic Programming, Princeton University Press, 1957.

[17] A. Bemporad, K. Fukuda, F. D. Torrisi, Convexity recognition of the union of polyhedra, Computational Geometry: Theory and Applications 18 (3) (2001) 141–154.

[18] F. Benoy, A. King, Inferring argument size relationships with CLP($\mathcal{R}$), in: J. P. Gallagher (ed.), Logic Program Synthesis and Transformation: Proceedings of the 6th International Workshop, vol. 1207 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Stockholm, Sweden, 1997.

[19] F. Besson, T. P. Jensen, J.-P. Talpin, Polyhedral analysis for synchronous languages, in: A. Cortesi, G. Filé (eds.), Static Analysis: Proceedings of the 6th International Symposium, vol. 1694 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Venice, Italy, 1999.

[20] G. Birkhoff, Lattice Theory, vol. XXV of Colloquium Publications, 3rd ed., American Mathematical Society, Providence, Rhode Island, USA, 1967.

[21] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), ACM Press, San Diego, California, USA, 2003.

[22] T. Bultan, R. Gerber, W. Pugh, Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results, ACM Transactions on Programming Languages and Systems 21 (4) (1999) 747–789.

[23] N. V. Chernikova, Algorithm for discovering the set of all solutions of a linear programming problem, U.S.S.R. Computational Mathematics and Mathematical Physics 8 (6) (1968) 282–293.

[24] M. A. Colón, H. B. Sipma, Synthesis of linear ranking functions, in: T. Margaria, W. Yi (eds.), Tools and Algorithms for Construction and Analysis of Systems, 7th International Conference, TACAS 2001, vol. 2031 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Genova, Italy, 2001.

[25] A. Cortesi, B. Le Charlier, P. Van Hentenryck, Combinations of abstract domains for logic programming: Open product and generic pattern construction, Science of Computer Programming 38 (1–3) (2000) 27–71.

[26] P. Cousot, Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming, in:

R. Cousot (ed.), Verification, Model Checking and Abstract Interpretation: Proceedings of the 6th International Conference (VMCAI 2005), vol. 3385 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Paris, France, 2005.

[27] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: B. Robinet (ed.), Proceedings of the Second International Symposium on Programming, Dunod, Paris, France, Paris, France, 1976.

[28] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, ACM Press, New York, 1979.

[29] P. Cousot, R. Cousot, Abstract interpretation frameworks, Journal of Logic and Computation 2 (4) (1992) 511–547.

[30] P. Cousot, R. Cousot, Inductive definitions, semantics and abstract interpretation, in: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press, Albuquerque, New Mexico, USA, 1992.

[31] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM Press, Tucson, Arizona, 1978.

[32] E. Davis, Constraint propagation with interval labels, Artificial Intelligence 32 (3) (1987) 281–331.

[33] G. Delzanno, A. Podelski, Model checking in CLP, in: R. Cleaveland (ed.), Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, vol. 1579 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Amsterdam, The Netherlands, 1999.

[34] D. Doose, Z. Mammeri, Polyhedra-based approach for incremental validation of real-time systems, in: L. T. Yang, M. Amamiya, Z. Liu, M. Guo, F. J. Rammig (eds.), Proceedings of the International Conference on Embedded and Ubiquitous Computing (EUC 2005), vol. 3824 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Nagasaki, Japan, 2005.

[35] N. Dor, M. Rodeh, S. Sagiv, Cleanness checking of string manipulations in C programs via integer analysis, in: P. Cousot (ed.), Static Analysis: 8th International Symposium, SAS 2001, vol. 2126 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Paris, France, 2001.

[36] L. Doyen, T. A. Henzinger, J.-F. Raskin, Automatic rectangular refinement of affine hybrid systems, in: P. Pettersson, W. Yi (eds.), Proceedings of the 3rd International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2005), vol. 3829 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Uppsala, Sweden, 2005.

[37] R. Ellenbogen, Fully automatic verification of absence of errors via interprocedural integer analysis, Master's thesis, School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel (Dec. 2004).

[38] G. Frehse, PHAVer: Algorithmic verification of hybrid systems past HyTech, in: M. Morari, L. Thiele (eds.), Hybrid Systems: Computation and Control: Proceedings of the 8th International Workshop (HSCC 2005), vol. 3414 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Zürich, Switzerland, 2005.

[39] G. Frehse, B. H. Krogh, R. A. Rutenbar, Verifying analog oscillator circuits using forward/backward refinement, in: Proceedings of the 9th Conference on Design, Automation and Test in Europe (DATE 06), ACM SIGDA, Munich, Germany, 2006, CD-ROM publication.

[40] K. Fukuda, A. Prodon, Double description method revisited, in: M. Deza, R. Euler, Y. Manoussakis (eds.), Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Brest, France, July 3-5, 1995, Selected Papers, vol. 1120 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996.

[41] D. Gopan, T. W. Reps, M. Sagiv, A framework for numeric analysis of array operations, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California, USA, 2005.

[42] P. Granger, Improving the results of static analyses programs by local decreasing iteration, in: R. K. Shyamasundar (ed.), Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science, vol. 652 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, New Delhi, India, 1992.

[43] P. Granger, Static analyses of congruence properties on rational numbers (extended abstract), in: P. Van Hentenryck (ed.), Static Analysis: Proceedings of the 4th International Symposium, vol. 1302 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Paris, France, 1997.

[44] B. S. Gulavani, S. K. Rajamani, Counterexample driven refinement for abstract interpretation, in: H. Hermanns, J. Palsberg (eds.), Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006), vol. 3920 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vienna, Austria, 2006.

[45] N. Halbwachs, Delay analysis in synchronous programs, in: C. Courcoubetis (ed.), Computer Aided Verification: Proceedings of the 5th International Conference, vol. 697 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Elounda, Greece, 1993.

[46] N. Halbwachs, D. Merchat, L. Gonnord, Some ways to reduce the space dimension in polyhedra computations, Formal Methods in System Design 29 (1) (2006) 79–95.

[47] N. Halbwachs, Y.-E. Proy, P. Roumanoff, Verification of real-time systems using linear relation analysis, Formal Methods in System Design 11 (2) (1997) 157–185.

[48] W. Hartong, L. Hedrich, E. Barke, On discrete modeling and model checking for nonlinear analog systems, in: E. Brinksma, K. G. Larsen (eds.), Computer Aided Verification: Proceedings of the 14th International Conference, vol. 2404 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Copenhagen, Denmark, 2002.

[49] T. A. Henzinger, The theory of hybrid automata, in: Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1996.

[50] T. A. Henzinger, P.-H. Ho, A note on abstract interpretation strategies for hybrid automata, in: P. J. Antsaklis, W. Kohn, A. Nerode, S. Sastry (eds.), Hybrid Systems II, vol. 999 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1995.

[51] T. A. Henzinger, P.-H. Ho, H. Wong-Toi, HyTech: A model checker for hybrid systems, Software Tools for Technology Transfer 1 (1+2) (1997) 110–122.

[52] T. A. Henzinger, J. Preussig, H. Wong-Toi, Some lessons from the hytech experience, in: Proceedings of the 40th Annual Conference on Decision and Control, IEEE Computer Society Press, 2001.

[53] C. Hymans, E. Upton, Static analysis of gated data dependence graphs, in: R. Giacobazzi (ed.), Static Analysis: Proceedings of the 11th International Symposium, vol. 3148 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Verona, Italy, 2004.

[54] G. Kahn, Natural semantics, in: F.-J. Brandenburg, G. Vidal-Naquet, M. Wirsing (eds.), Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, vol. 247 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Passau, Germany, 1987.

[55] M. Karr, Affine relationships among variables of a program, Acta Informatica 6 (1976) 133–151.

[56] K. Krishnan, J. Mitchell, A unifying framework for several cutting plane methods for semidefinite programming, Optimization Methods and Software 21 (1) (2006) 57–74.

[57] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Automating mimicry attacks using static binary analysis, in: Proceedings of Security '05, the 14th USENIX Security Symposium, Baltimore, MD, USA, 2005.

[58] J.-L. Lassez, M. J. Maher, On Fourier's algorithm for linear arithmetic constraints, J. Autom. Reasoning 9 (3) (1992) 373–379.

[59] H. Le Verge, A note on Chernikova's algorithm, *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France (1992).

[60] V. Loechner, *PolyLib*: A library for manipulating parameterized polyhedra, Available at `http://icps.u-strasbg.fr/~loechner/polylib/`, declares itself to be a continuation of [79] (Mar. 1999).

[61] F. Mesnard, R. Bagnara, cTI: A constraint-based termination inference tool for ISO-Prolog, Theory and Practice of Logic Programming 5 (1&2) (2005) 243–257.

[62] A. Miné, Weakly relational numerical abstract domains, Ph.D. thesis, École Polytechnique, Paris, France (Mar. 2005).

[63] T. S. Motzkin, H. Raiffa, G. L. Thompson, R. M. Thrall, The double description method, in: H. W. Kuhn, A. W. Tucker (eds.), Contributions to the Theory of Games – Volume II, No. 28 in Annals of Mathematics Studies, Princeton University Press, Princeton, New Jersey, 1953, pp. 51–73.

[64] T. Nakanishi, K. Joe, C. D. Polychronopoulos, A. Fukuda, The modulo interval: A simple and practical representation for program analysis, in: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Newport Beach, California, USA, 1999.

[65] S. P. K. Nookala, T. Risset, A library for Z-polyhedral operations, *Publication interne* 1330, IRISA, Campus de Beaulieu, Rennes, France (2000).

[66] G. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Computer Science Department, University of Aarhus, Denmark (1981).

[67] P. Quinton, S. Rajopadhye, T. Risset, On manipulating Z-polyhedra, Tech. Rep. 1016, IRISA, Campus Universitaire de Bealieu, Rennes, France (Jul. 1996).

[68] S. Sankaranarayanan, M. Colón, H. B. Sipma, Z. Manna, Efficient strongly relational polyhedral analysis, in: E. A. Emerson, K. S. Namjoshi (eds.), Verification, Model Checking and Abstract Interpretation: Proceedings of the 7th International Conference (VMCAI 2006), vol. 3855 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Charleston, SC, USA, 2006.

[69] S. Sankaranarayanan, H. B. Sipma, Z. Manna, Scalable analysis of linear systems using mathematical programming, in: R. Cousot (ed.), Verification, Model Checking and Abstract Interpretation: Proceedings of the 6th International Conference (VMCAI 2005), vol. 3385 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Paris, France, 2005.

[70] D. A. Schmidt, Natural-semantics-based abstract interpretation (preliminary version), in: A. Mycroft (ed.), Static Analysis: Proceedings of the 2nd International Symposium, vol. 983 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Glasgow, UK, 1995.

[71] A. Schrijver, Theory of Linear and Integer Programming, Wiley Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, 1999.

[72] R. Shaham, E. K. Kolodner, S. Sagiv, Automatic removal of array memory leaks in Java, in: D. A. Watt (ed.), Proceedings of the 9th International Conference on Compiler Construction (CC 2000), vol. 1781 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Berlin, Germany, 2000.

[73] A. Simon, A. King, J. M. Howe, Two variables per linear inequality as an abstract domain, in: M. Leuschel (ed.), Logic Based Program Synthesis and Tranformation, 12th International Workshop, vol. 2664 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Madrid, Spain, 2002.

[74] K. Sohn, A. Van Gelder, Termination detection in logic programs using argument sizes (extended abstract), in: Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, Association for Computing Machinery, Denver, Colorado, United States, 1991.

[75] H. Song, K. J. Compton, W. C. Rounds, SPHIN: a model checker for reconfigurable hybrid systems based on SPIN, in: R. Lazic, R. Nagarajan (eds.), Proceedings of the 5th International Workshop on Automated Verification of Critical Systems, vol. 145 of Electronic Notes in Theoretical Computer Science, University of Warwick, UK, 2006.

[76] J. Stoer, C. Witzgall, Convexity and Optimization in Finite Dimensions I, Springer-Verlag, Berlin, 1970.

[77] K. van Hee, O. Oanea, N. Sidorova, M. Voorhoeve, Verifying generalized soundness for workflow nets, in: I. Virbitskaite, A. Voronkov (eds.), Perspectives of System Informatics: Proceedings of the Sixth International Andrei Ershov Memorial Conference, vol. 4378 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Akademgorodok, Novosibirsk, Russia, 2006.

[78] A. Venet, G. Brat, Precise and efficient static array bound checking for large embedded C programs, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04), ACM Press, Washington, DC, USA, 2004.

[79] D. K. Wilde, A library for doing polyhedral operations, Master's thesis, Oregon State University, Corvallis, Oregon, also published as IRISA *Publication interne* 785, Rennes, France, 1993 (Dec. 1993).