Incremental Invariant Generation using Logic-based Automatic Abstract Transformers*

Pierre-Loïc Garoche[1,2], Temesghen Kahsai[2] and Cesare Tinelli[2]

[1] Onera, the French Aerospace Lab, France
[2] The University of Iowa

Abstract. Formal analysis tools for system models often require or benefit from the availability of auxiliary system invariants. Abstract interpretation is currently one of the best approaches for discovering useful invariants, in particular numerical ones. However, its application is limited by two orthogonal issues: (*i*) developing an abstract interpretation is often non-trivial; each transfer function of the system has to be represented at the abstract level, depending on the abstract domain used; (*ii*) with precise but costly abstract domains, the information computed by the abstract interpreter can be used only once a post fix point has been reached; this may take a long time for large systems or when widening is delayed to improve precision. We propose a new, completely automatic, method to build abstract interpreters which, in addition, can provide sound invariants of the system under analysis before reaching the end of the post fix point computation. In effect, such interpreters act as on-the-fly invariant generators and can be used by other tools such as logic-based model checkers. We present some experimental results that provide initial evidence of the practical usefulness of our method.

1 Introduction and Motivation

Abstract interpretation and symbolic model checking have led independently over the years to the creation of analysis tools that are starting to have a substantial impact on the development of real world software, in particular for safety- or mission-critical systems. Interestingly, the two exhibit complementary strengths and weaknesses [13]. Model checking techniques so far have proved stronger on software that is mostly control-driven and not heavily data-dependent. To be effective with data-dependent programs, these techniques may require programs to be judiciously annotated with data invariants. Also, model checking has been traditionally limited to finite-state systems although new approaches, such as those based on solvers for Satisfiability Modulo Theories (SMT), can lift that restriction in some cases.

Dually, abstract interpretation techniques are quite effective with data-dependent programs, in particular numerical ones, requiring in principle no program annotations. On the other hand, they have more difficulties in dealing with control aspects [13]. Also, although abstract interpretation is a very general framework, most of its applications focus on the analysis of source code. Even tools, such as Nbac [16], that target software

*Work supported by AFOSR grant #AF9550-09-1-0517, FNRAE Cavale project and ANR INS Project CAFEIN, with the support of the Aerospace Space cluster.

artifacts at a higher level of abstraction (e.g., software models expressed in dataflow specification languages) do not analyze those artifacts directly and work instead with their compilation into an intermediate imperative representation such as LLVM or byte code. This is possibly a consequence of the fact that developing an abstract interpreter for a complete language can be time consuming: even if a large set of abstract domains, such as those provided by the APRON library [17], is readily available, defining sound abstract transformers for every construct of the target language requires substantial work. Another limitation of current abstract interpretation techniques is that they typically depend on Kleene-style fix point algorithms to construct an abstract semantics of the program under analysis. The properties of such semantics, characterized by the concretization of a post fix point of an abstract transformer, can be obtained only once the post fix point has been (completely) computed. Depending on the widening strategies used or, in general, the complexity of the abstractions and the semantics considered, one may have to wait a long time before getting any interesting information from the analysis of the program.

Contribution and significance In this work we try to address some of the issues above by combining techniques from abstract interpretation and logic-based model checking. Specifically, we propose a general method for the automatic definition of abstract interpreters that compute numerical invariants of transition systems. We rely on the possibility of encoding the transition system in a decidable logic to compute transformers for an abstract interpreter *completely automatically*. Our method has the significant added benefit that the abstract interpreter can be instrumented to generate system invariants on the fly, during its iterative computation of a post fix point. A prototype implementation of the method provides initial evidence of the feasibility of our approach.

While motivated by practical issues (namely, the generation of auxiliary invariants for a k-induction model checker) the current work is more general and can be adapted to a wide variety of contexts. It only requires that the transition system semantics be expressible in a decidable logic with an efficient solver, such as SAT or SMT solvers, and that the elements of the chosen abstract domain be effectively representable in that logic, as discussed later in more detail. Such requirements are satisfied by a large number of abstract domains used in current practice. As a consequence, we believe that our approach could help considerably in expanding the reach of abstract interpretation techniques to a variety of target languages, as well as facilitate their integration with complementary techniques from model checking.

Related work With the current efficiency of SMT solvers on the one hand and the ability of abstract interpretation to compute numerical invariants on the other, the issue of combining SMT and Abstract Interpretation is receiving increasing attention. In [7], Cousot *et al,* draw a parallel between SMT-based reasoning and abstract interpretation. They identify the Nelson-Oppen procedure as a reduced product over different interpretations. While this work is more general, it allows one to understand ours as follows: the concrete domain is an abstract logical domain, our concrete transformer—computed with the aid of an SMT solver—can be seen as an over-approximation of the concrete transition relation in this abstract logical domain. The abstraction we build amounts to computing a reduction between a logical and an algebraic domain, as suggested in [7, §6]. Comparable work in [30], gives an overview of techniques embedding logical pred-

icates as elements of *logical lattices*. Some SMT theories are then formalized within this abstract interpretation view of the analysis: uninterpreted function symbols, linear arithmetic, and their combination.

Another, more practical approach by Monniaux and Gonnord [23] uses bounded reachability with an SMT solver to compute a chaotic iteration strategy. The solver identifies the equation that needs propagating in order to achieve a better widening. However, unlike ours, this solution does not use the actual models found by the SMT solver. In [10], an SMT solver is used to choose among different strategies in an iteration-based policy analysis. The solver identifies the next strategy that will improve the current abstract property. While both works rely on SMT solvers to aid the fix point computation, they do not encode, as we do, the concrete transition relation as a SMT formula in order to compute the abstract property. Also related is Monniaux's automatic modular abstraction for linear constraints [22]. A predicate transformer is defined using quantifier elimination over the semantics of C statements, as in an axiomatic semantics (weakest precondition or strongest postcondition). The transformer is exact for the linear template abstractions considered. It is unclear, however, how this approach can scale to a complete program analysis, since the use of quantifier elimination on a complete transition system is not usually feasible (the blocks analyzed in [22] are small functions used in a symbol library for Lustre/Scade). King and Sondergaard [21] follow a similar approach but rely on reasoning in a concrete logic, as we do, and then abstract the result. As in [22], they aim at computing a very precise transformer but restrict themselves to a specific setting with finite domains, whereas we do not.

A line of work by Reps and various collaborators [24, 31, 28, 29] shares similar foundations with our approach: relying on a decidable logic to construct abstract transformers. The work in [24] over-approximates least fix points but is restricted to domains admitting only finite height chains, while that in [31] adopts the dual approach—to avoid the convergence issue in infinite height domains—by over-approximating greatest fix points from above. Both works are based on manually defined abstract transformers. Very recent work [28, 29], concurrent with ours, extends those approaches by synthesizing automatically the abstract transformer via a logic encoding, in a way similar to ours. The first paper combines a least and a greatest fix point computation, while the second only relies on a greatest fix point over-approximation. In case of infinite height domains (e.g., intervals or polyhedra), the least fix point approximation will never converge and only the greatest fix point may be used. In contrast, we target the over-approximation of the least fix point, using widening to ensure convergence. Regarding the ability to produce safe abstract values before the end of the fix point computation, it is not clear how a greatest fix point approach would compare in practice to our incremental invariant generation mechanism (cf. Section 4). A comparative experimental evaluation would require a substantial effort that is outside the scope of this paper.

Finally, the static analysis tools to which we compare ours experimentally in Section 5 are based on sophisticated techniques to improve the precision of the fix point computation, such as lookahead widening [12], or to accelerate convergence [11, 26]. These techniques, as well as others such as delayed widening could be integrated in principle in our approach since they mainly focus on the iteration strategy for the fix point computation rather than a specific abstract transformer.

2 Formal Preliminaries

We use basic notions and results from abstract interpretation (e.g. [4, 5]). We introduce below those that are most relevant to this work, to have a more self-contained presentation. Similarly, we also introduce relevant notions from symbolic logic and automated reasoning. As customary, we model computational systems as transition systems. A *transition system* S is a triple $(\mathbf{Q}, \mathbf{I}, \rightsquigarrow)$ where $\mathbf{Q}$ is a set of *states*, the *state space*; $\mathbf{I} \subseteq \mathbf{Q}$ is the set of S's *initial states*; and $\rightsquigarrow \subseteq \mathbf{Q} \times \mathbf{Q}$ is S's *transition relation*. A state $q' \in \mathbf{Q}$ is *reachable* if $q' \in \mathbf{I}$ or $q \rightsquigarrow q'$ for some reachable state q.

Abstract Interpretation Abstract interpretation allows one to analyze a transition system $S = (\mathbf{Q}, \mathbf{I}, \rightsquigarrow)$ by first defining a *concrete domain* for S, a partially ordered set $\langle D, \subseteq \rangle$, and a *concrete transformer*, a monotonic function $f : D \to D$. In this paper we will focus on the *collecting semantics*

$$\mathbb{S} \stackrel{\text{def}}{=} \mathrm{lfp}^{\subseteq}_{\mathbf{I}}(f)$$

of S where $D = \wp(\mathbf{Q})$, the power set of $\mathbf{Q}$; $\subseteq$ is set inclusion; $f(X) = X \cup \{x' \mid x \in X,\ x \rightsquigarrow x'\}$; and $\mathrm{lfp}^{\subseteq}_{\mathbf{I}}(f)$ is the least-fix point of f greater than $\mathbf{I}$, obtained as the stationary limit of the ascending sequence $X_0 \subseteq X_1 \subseteq \ldots$ with $X_0 = \mathbf{I}$ and $X_n = f(X_{n-1})$ for all $n > 0$.

An abstract representation of the concrete domain is provided by another partial order $\langle D^\#, \sqsubseteq^\# \rangle$ the *abstract domain*. The two are related by an *abstraction function* $\alpha : D \mapsto D^\#$ and a *concretization function* $\gamma : D^\# \mapsto D$. An *abstract transformer* is any monotonic function $g : D^\# \to D^\#$. We will consider domains $\langle D, \subseteq \rangle$ and $\langle D^\#, \sqsubseteq \rangle$ that are lattices, and abstraction and concretization functions that form a *Galois connection* (which we denote by $\alpha : \langle D, \subseteq \rangle \leftrightarrows \langle D^\#, \sqsubseteq \rangle : \gamma$). In a Galois connection, both α and γ are monotonic; $\alpha(\gamma(y)) \sqsubseteq y$ for all $y \in D^\#$; and $x \subseteq \gamma(\alpha(x))$ for all $x \in D$.

First-order logic Our method works with several logics (including propositional and quantified Boolean logic) that can be more or less directly embedded in many-sorted first-order logic with equality (e.g. [8]). For generality then, we present our work in terms of the latter. We fix a set $\mathbf{S}$ of *sort symbols* and let $\mathbf{X} = \bigcup_{\sigma \in \mathbf{S}} \mathbf{X}_\sigma$ where each $\mathbf{X}_\sigma$ is an infinite set of *variables (of sort σ)*. Given a many-sorted signature Σ of function and predicate symbols, well-sorted terms and formulas (resp. *Σ-terms* and *Σ-formulas*) are defined as usual. If F is a Σ-formula, and $\mathbf{x} = (x_1, \ldots, x_n)$ a tuple of variables with no repetitions, we write $F[\mathbf{x}]$ to denote that F's free variables are from $\mathbf{x}$; furthermore, if $\mathbf{t} = (t_1, \ldots, t_n)$ is a term tuple, we write $F[\mathbf{t}]$ to denote the formula obtained from F by simultaneously replacing each occurrence of x_i in F by t_i for $i = 1, \ldots, n$.

We adopt a standard notion of *Σ-interpretation* $\mathcal{M}$ for each signature Σ. A satisfiability relation $\models$ between such interpretations and Σ-formulas with variables in $\mathbf{X}$ is defined inductively as usual. A Σ-interpretation $\mathcal{M}$ *satisfies* a Σ-formula F if $\mathcal{M} \models F$. We are normally interested in specific classes of Σ-formulas and Σ-interpretations. We collect these restrictions in the notion of a *(sub)logic* (of many-sorted logic): a triple $\mathcal{L} = (\Sigma, \mathbf{F}, \mathbf{M})$ where Σ is a signature; $\mathbf{F}$, the *language* of $\mathcal{L}$, is a set of Σ-formulas; and $\mathbf{M}$ is a class of Σ-interpretations, the *models* of $\mathcal{L}$, that is closed under variable reassignment, (i.e., every Σ-interpretation that differs from one in $\mathbf{M}$ only for how it interprets the variables is also in $\mathbf{M}$). A formula $F[\mathbf{x}]$ of $\mathcal{L}$ is *satisfiable (resp., unsatisfiable) in $\mathcal{L}$*

if it is satisfied by some (resp., no) interpretation in $\mathbf{M}$. A set Γ of formulas *entails in* $\mathcal{L}$ a Σ-formula F, written $\Gamma \models_{\mathcal{L}} F$, if $\Gamma \cup \{F\} \in \mathbf{F}$ and every interpretation in $\mathbf{M}$ that satisfies all formulas in Γ satisfies F as well. The set Γ is *satisfiable in* $\mathcal{L}$ if $\Gamma \not\models_{\mathcal{L}} \mathsf{false}$.

3 Computable abstract transformer via logic encodings

For the rest of the paper we fix a transition system $S = (\mathbf{Q}, \mathbf{I}, \rightsquigarrow)$ and its collecting semantics $\mathbb{S} = \mathrm{lfp}^{\subseteq}_{\mathbf{I}}(f)$ introduced earlier, which coincides with the set of reachable states of S. Our main concern will be how to define a sound abstract counterpart $f_{\mathbf{A}}$ of f in a suitable abstract domain $\langle \mathbf{A}, \sqsubseteq_{\mathbf{A}} \rangle$ with abstraction function $\alpha : \wp(\mathbf{Q}) \to \mathbf{A}$ and concretization function $\gamma : \mathbf{A} \to \wp(\mathbf{Q})$ so that we can define S's abstract semantics as

$$\mathbb{S}^{\#} \stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq_{\mathbf{A}}}_{\mathbf{I}_{\mathbf{A}}}(f_{\mathbf{A}})$$

where $\mathbf{I}_{\mathbf{A}}$ is in turn a suitable abstraction of $\mathbf{I}$. By well-known results [4, 5], the fix point $\mathbb{S}^{\#}$ above can be computed or over-approximated so that its concretization by γ is a sound approximation (i.e., an over-approximation) of the concrete fix point $\mathbb{S}$.

A major issue when using abstract interpretation in general is how to define $f_{\mathbf{A}}$. In practice, when the transition system is induced by a program, as is often the case, the concrete transformer f is defined constructively in terms of the programming language's idioms (e.g., assignment, loop and conditional statements for imperative languages) and memory model (e.g., heap, stack, etc.). The corresponding abstract transformer must then handle all those constructs as well, and reflect their respective actions in the abstract domain $D_{\mathbf{A}}$. When the abstraction function α is defined from γ by the unique adjoint property of Galois connections the definition of $f_{\mathbf{A}}$ is usually a manual, laborious chore. One has to design the transformer in detail and then prove it sound, by showing that $f(X) \subseteq \gamma(f_{\mathbf{A}}(a))$ for all $a \in \mathbf{A}$ and $X \subseteq \gamma(a)$.

We present a method that can instead compute a sound abstraction of f completely automatically. The method is applicable when the transition system and the concrete and abstract domains can be encoded as we explain below in a logic $\mathcal{L}$ satisfying the requirements listed in the next subsection. For generality, we will describe our method in terms of an arbitrary logic $\mathcal{L}$ satisfying those requirements. To have an intuition, however, depending on the concrete domain, possible examples of $\mathcal{L}$ would be propositional logic or several of the many logics used in SMT: linear real arithmetic, linear integer arithmetic with arrays, and so on.

The basic idea of our method for computing the abstract transformer is fairly simple. It depends on the availability of a $\mathcal{L}$-formula T encoding S's transition relation and a computable function $\gamma_{\mathbf{F}}$ mapping each abstract element a to a formula $\gamma_{\mathbf{F}}(a)$ satisfied by the states abstracted by a. Given an $a \in \mathbf{A}$, the transformer uses T, $\gamma_{\mathbf{F}}(a)$ and a solver for $\mathcal{L}$ to look for a state $\mathbf{v}'$ that is not abstracted by a but is the successor of a state abstracted by a. If such a state does not exist then a is a fix point and is returned. Otherwise, the transformer computes an abstraction a' of state $\mathbf{v}'$ and returns the join of a and a'.

The main appeal of this approach is that logic solvers enumerating satisfying assignments are readily available, and abstracting single states is straightforward for most abstract domains used in practice. In principle, a better approach would be to compute

not a single state like $\mathbf{v}'$ above but a formula G denoting a whole set of them. The resulting abstract transformer would then require a smaller number of iterations to reach a fix point. This would both accelerate convergence and, since we use widening, improve precision by possibly needing fewer widening steps. However, computing the formula G and mapping it to a corresponding abstract element is considerably more challenging and expensive, if possible at all for a chosen logic and abstract domain. So we leave the investigation of this approach to further work. The rest of this section formalizes our current approach and describes it in more detail.

Logic requirements. We assume a logic $\mathcal{L} = (\Sigma, \mathbf{F}, \mathbf{M})$ with a decidable entailment relation $\models_{\mathcal{L}}$ and a language $\mathbf{F}$ closed under all the Boolean operators.[3] For each sort σ in $\mathcal{L}$, we distinguish a set V_σ of variable-free terms, which we call *values*, such that $\models_{\mathcal{L}} \neg(v_1 = v_2)$ for each distinct $v_1, v_2 \in V_\sigma$. Examples of values would be Boolean, integer or rational constants. We assume that the satisfiable formulas of $\mathcal{L}$ are *satisfied by values*, that is, for every formula $F[\mathbf{y}]$ (with free variables from $\mathbf{y}$) satisfied by a model $\mathcal{M}$ of $\mathcal{L}$ there is a value tuple $\mathbf{v}$ such that $F[\mathbf{v}]$ is satisfied by $\mathcal{M}$.

We assume a total surjective encoding of S's state space $\mathbf{Q}$ to n-tuples of values, for some fixed n, where each n-tuple encodes a state. Depending on $\mathcal{L}$, states may be encoded, for instance, as tuples of Boolean constants, or integer constants, or mixed tuples of Boolean, integer and rational constants, and so on. From now on then *we will identify states with tuples of values*. Note that, thanks to our various assumptions, each formula $F[\mathbf{y}_1, \ldots, \mathbf{y}_k]$ in $k \cdot n$ variables denotes a subset of $\mathbf{Q}^k$, namely the set of all k-tuples of states that satisfy F. We call that set the *extension* of F and define it formally as follows: $[\![F]\!] \stackrel{\text{def}}{=} \{(\mathbf{v}_1, \ldots, \mathbf{v}_k) \in \mathbf{Q}^k \mid F[\mathbf{v}_1, \ldots, \mathbf{v}_k] \text{ is satisfiable in } \mathcal{L}\}$. We refer to formulas like F above as *state formulas* and say they are *satisfied* by the state tuples in $[\![F]\!]$. For each state $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbf{Q}$ and distinct variables $\mathbf{x} = (x_1, \ldots, x_n)$ of corresponding sort, we denote by $A_{\mathbf{v}}$ the *assignment formula* $x_1 = v_1 \wedge \cdots \wedge x_n = v_n$, which is satisfied exactly by $\mathbf{v}$. Finally, we assume the existence of an *encoding of S in $\mathcal{L}$*, i.e., a pair $(I[\mathbf{x}], T[\mathbf{x}, \mathbf{x}'])$ of formulas of $\mathcal{L}$ with $\mathbf{x}$ and $\mathbf{x}'$ both of size n, where $I[\mathbf{x}]$ is a formula satisfied exactly by the initial states of S, and $T[\mathbf{x}, \mathbf{x}']$ is a formula satisfied by two reachable states $\mathbf{v}, \mathbf{v}'$ iff $\mathbf{v} \rightsquigarrow \mathbf{v}'$.

First abstraction—from sets of states to formulas For theoretical convenience, we start with an intermediate abstraction that maps sets of states to possibly infinitary formulas representing those states precisely. To do that, we extend the language of $\mathcal{L}$ by closing it under a disjunction operator $\bigvee$ that applies to (possibly infinite) sets of formulas of $\mathcal{L}$. We then extend the notions of satisfiability, entailment and equivalence in $\mathcal{L}$ to the new language as expected—e.g., for every set Γ of formulas of $\mathcal{L}$, $\bigvee \Gamma$ is satisfied by an interpretation $\mathcal{M}$ if some $F \in \Gamma$ is satisfied by $\mathcal{M}$, and so on.[4]

Let $\mathbf{F}_{\mathbf{x}}$ be the set of all formulas in the extended language above whose free variables are from the same n-tuple $\mathbf{x}$. One can show that mutual entailment between two formulas in $\mathbf{F}_{\mathbf{x}}$ is an equivalence relation. Let $[F]$ denote the equivalence class of a formula F with respect to this relation, and let $\mathbf{E}$ denote the set of all those equivalence classes. Let $[\![[F]]\!] \stackrel{\text{def}}{=} [\![F]\!]$ for each $[F] \in \mathbf{E}$. The poset $\langle \mathbf{E}, \sqsubseteq_{\mathbf{E}} \rangle$ where

[3] The latter is to simplify the exposition. Weaker assumptions are possible.

[4] In practice, our method will never need to work with formulas $\bigvee \Gamma$ where Γ is infinite.

$$[F] \sqsubseteq_{\mathbf{E}} [G] \quad \text{iff} \quad F \models_{\mathcal{L}} G$$

has a lattice structure with the following join and meet operators: $[F] \sqcup_{\mathbf{E}} [G] \stackrel{\text{def}}{=} [F \vee G]$ and $[F] \sqcap_{\mathbf{E}} [G] \stackrel{\text{def}}{=} [F \wedge G]$. It can be shown that the two functions[5]

$$\alpha_{\mathbf{E}} : \wp(\mathbf{Q}) \to \mathbf{E} \stackrel{\text{def}}{=} \lambda V.\, [\bigvee \{A_{\mathbf{v}} \mid \mathbf{v} \in V\}] \qquad \text{and} \qquad \gamma_{\mathbf{E}} : \mathbf{E} \to \wp(\mathbf{Q}) \stackrel{\text{def}}{=} \lambda E.\, [\![E]\!]$$

form a Galois connection. By standard results [5], the best sound abstract transformer of f with respect to this connection is

$$f_{\mathbf{E}} : \mathbf{E} \to \mathbf{E} \stackrel{\text{def}}{=} \alpha_{\mathbf{E}} \circ f \circ \gamma_{\mathbf{E}} = \lambda E.\, [\bigvee \{A_{\mathbf{v}} \mid \mathbf{v} \in [\![E]\!] \cup \{\mathbf{u}' \mid \mathbf{u} \in [\![E]\!],\, \mathbf{u} \rightsquigarrow \mathbf{u}'\}\}]$$

By our logic requirements, the most precise abstraction of the set $\mathbf{I}$ of S's initial states is $\alpha_{\mathbf{E}}(\mathbf{I}) = [I]$ where, recall, I is the formula denoting $\mathbf{I}$ in $\mathcal{L}$. It follows that in the abstract domain $\langle \mathbf{E}, \sqsubseteq_{\mathbf{E}} \rangle$ we can define the following semantics for S: $\mathbb{S}^{\mathbf{E}} \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(f_{\mathbf{E}})$.

Second abstraction—changing fix point computation For our later needs, we would like to have a fix point computation that actually enumerates the additional states discovered by the collecting semantics. The abstraction $\alpha_{\mathbf{E}}$ above, over-approximating sets of states by disjunctions of assignment formulas, is not well suited for that because these disjunctions can be infinitary. Hence, we introduce another abstract transformer, on the same lattice $\langle \mathbf{E}, \sqsubseteq_{\mathbf{E}} \rangle$:

$$g_{\mathbf{E}} : \mathbf{E} \to \mathbf{E} \stackrel{\text{def}}{=} \lambda E.\, E \sqcup_{\mathbf{E}} \mathsf{choose}(\{[A_{\mathbf{v}'}] \mid T[\mathbf{v}, \mathbf{v}'] \text{ is sat. in } \mathcal{L},\, \mathbf{v} \in [\![E]\!],\, \mathbf{v}' \notin [\![E]\!]\})$$

where choose is some choice function over subsets of $\mathbf{E}$, returning one element of its input set if the set is non-empty, and [false] otherwise. This function maps each equivalence class E to a class E' such that $[\![E']\!] \setminus [\![E]\!]$ contains just one state, chosen among the successors of the states in $[\![E]\!]$ according to the transition formula T. We can use $g_{\mathbf{E}}$ instead of $f_{\mathbf{E}}$ in the fix point computation thanks to the following result.[6]

Proposition 1 (Soundness). *The transformers $f_{\mathbf{E}}$ and $g_{\mathbf{E}}$ have the same least fix point above $[I]$, i.e., $\text{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(f_{\mathbf{E}}) = \text{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(g_{\mathbf{E}})$ where $\text{lfp}^{\sqsubseteq_{\mathbf{E}}}$ is defined using transfinite iterations.*

Main abstraction—abstracting formulas in $\mathbf{F}_{\mathbf{x}}$ We now introduce our last abstraction, mapping formulas in $\mathbf{F}_{\mathbf{x}}$ to elements of an abstract domain $\langle \mathbf{A}, \sqsubseteq_{\mathbf{A}} \rangle$ like those typically used in abstract interpretation tools (intervals, polyhedra, and so on). We assume that $\mathbf{A}$ is fitted with a lattice structure with meet $\sqcap_{\mathbf{A}}$ and join $\sqcup_{\mathbf{A}}$. We also assume the existence of a *computable* monotonic function $\gamma_{\mathbf{F}} : \mathbf{A} \to \mathbf{F}_{\mathbf{x}}$ that associates a formula of $\mathbf{F}_{\mathbf{x}}$ to each element of $\mathbf{A}$. Intuitively, we are requiring that each element of $\mathbf{A}$ be effectively representable as a formula denoting a set of states. This requirement is easily satisfied for many numerical abstract domains and the sort of logics used in SMT. For instance, intervals can be mapped to conjunctions of inequalities between variables

[5] We borrow λ-calculus' notation to denote mathematical functions.

[6] All proofs of our results can be found in a companion technical report available at http://www.cs.uiowa.edu/~tinelli/html/publications.html.

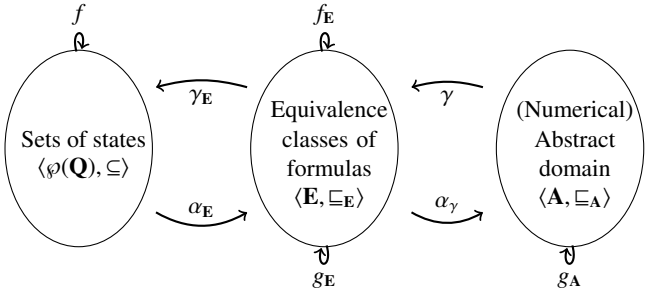

Fig. 1: Global framework: combination of abstractions.

Input: $a \in \mathbf{A}$
$F[\mathbf{x}, \mathbf{x}'] := \gamma_{\mathbf{F}}(a)[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge \neg\gamma_{\mathbf{F}}(a)[\mathbf{x}']$
if F is satisfiable in $\mathcal{L}$ **then**
  let $\mathbf{v}, \mathbf{v}'$ be two states that satisfy $F[\mathbf{x}, \mathbf{x}']$
  return $a \sqcup_{\mathbf{A}} \alpha_{\mathbf{Q}}(\mathbf{v}')$
return a

Fig. 2: Basic version of the automatic abstract transformer $g_{\mathbf{A}}$.

and values; similarly, any linear-based abstraction can be mapped to a conjunction of linear arithmetic constraints. As concretization function we use the monotonic function $\gamma : \mathbf{A} \mapsto \mathbf{E} \stackrel{\text{def}}{=} (\lambda F.[F]) \circ \gamma_{\mathbf{F}}$ which maps each abstract element to an equivalence class in $\mathbf{E}$. Since $\mathbf{E}$ and $\mathbf{A}$ are lattices, γ induces a Galois connection $\alpha_\gamma : \langle \mathbf{E}, \sqsubseteq_{\mathbf{E}} \rangle \leftrightarrows \langle \mathbf{A}, \sqsubseteq_{\mathbf{A}} \rangle : \gamma$ where α_γ is uniquely determined by γ.

In summary, we obtain the combination of abstractions illustrated in Figure 1. However, *we do not use α_γ at all* by assuming instead the existence of a *state abstraction* function $\alpha_{\mathbf{Q}} : \mathbf{Q} \mapsto \mathbf{A}$ which directly associates states to their abstract counterparts in $\mathbf{A}$. For our approach to be sound, it is enough for $\alpha_{\mathbf{Q}}$ to be such that $a \sqsubseteq_{\mathbf{A}} \alpha_{\mathbf{Q}}(\mathbf{v})$ for each $\mathbf{v} \in \mathbf{Q}$ and $a \in \mathbf{A}$ where a is $\mathbf{v}$'s best abstraction—i.e., the smallest element of $\mathbf{A}$ with $[A_{\mathbf{v}}] \sqsubseteq_{\mathbf{E}} \gamma(a)$. In the actual domains we have considered in our implementation, the definition of $\alpha_{\mathbf{Q}}$ is straightforward and such that $a = \alpha_{\mathbf{Q}}(\mathbf{v})$. For instance, let $\mathbf{v} = (4, -2, 5)$ Then $\alpha_{\mathbf{Q}}(\mathbf{v})$ is $([4; 4], [-2; -2], [5; 5])$ if $\mathbf{A}$ is the integer interval domain, and is the abstract element described by the system $\{4 \leq x_1 \leq 4,\ -2 \leq x_2 \leq -2,\ 5 \leq x_3 \leq 5\}$ if $\mathbf{A}$ is a relational domain such as octagons or polyhedra.

The abstract transformer Recall that our main goal was to generate a computable sound abstract transformer $g_{\mathbf{A}}$ for $g_{\mathbf{E}}$ *automatically*. We can do that by relying solely on (*i*) the function $\gamma_{\mathbf{F}}$, (*ii*) the state abstraction $\alpha_{\mathbf{Q}}$, and (*iii*) a sound, complete and terminating satisfiability solver for the logic $\mathcal{L}$ that is also able to return for each satisfiable state formula $F[\mathbf{x}_1, \ldots, \mathbf{x}_k]$ a tuple $\mathbf{v}_1, \ldots, \mathbf{v}_k$ of states that satisfies it.

A basic procedure for computing $g_{\mathbf{A}}$ is given in Figure 2. The satisfiability tests and the choice of the states $\mathbf{v}$ and $\mathbf{v}'$ in the figure are performed by the solver for $\mathcal{L}$, which effectively plays for $g_{\mathbf{A}}$ the role of the choice function in the definition of $g_{\mathbf{E}}$. We note that, while fix points are traditionally computed in the abstract domain, with our approach it is not necessary to transfer back the element $g_{\mathbf{A}}(a)$ to detect that a is a fix point: it is enough to detect that the formula F in Figure 2 is unsatisfiable.

Theorem 1 (Soundess). *The transformer $g_{\mathbf{A}}$ is a sound approximation of $g_{\mathbf{E}}$: for all $a \in \mathbf{A}$, $(g_{\mathbf{E}} \circ \gamma)(a) \sqsubseteq_{\mathbf{E}} (\gamma \circ g_{\mathbf{A}})(a)$.*

Our eventual goal is to over-approximate the fix point $\mathrm{lfp}^{\sqsubseteq_{\mathbf{A}}}_{I_{\mathbf{A}}}(g_{\mathbf{A}})$ where $I_{\mathbf{A}}$ is a sound approximation of the initial state formula I; more precisely, where $[I] \sqsubseteq_{\mathbf{E}} \gamma(I_{\mathbf{A}})$. When I is satisfied by a single state $\mathbf{v}$, the abstract element $I_{\mathbf{A}}$ is just $\alpha_{\mathbf{Q}}(\mathbf{v})$. In general, we can use the logic solver again to compute an $I_{\mathbf{A}}$ iteratively. A basic procedure for that (also used in [24]) is the following, starting with $I_{\mathbf{A}}$ equal to the bottom element of $\mathbf{A}$:

while (there is a state $\mathbf{v}$ satisfying $I[\mathbf{x}] \wedge \neg\gamma_{\mathbf{F}}(I_{\mathbf{A}})[\mathbf{x}]$) **do**
  $I_{\mathbf{A}} := I_{\mathbf{A}} \sqcup_{\mathbf{A}} \alpha_{\mathbf{Q}}(\mathbf{v})$

Proposition 2 (Soundness). *When the loop above terminates, the computed element $I_{\mathbf{A}}$ is a sound approximation of $[I]$.*

In practice, we are mostly interested in abstract domains that do not satisfy the ascending chain condition [4]. In those cases, a widening operator ∇ is needed in addition to the join $\sqcup_{\mathbf{A}}$, in the computation of $I_{\mathbf{A}}$ and of $\mathrm{lfp}^{\sqsubseteq_{\mathbf{A}}}_{I_{\mathbf{A}}}(g_{\mathbf{A}})$ to ensure convergence. Although any of the widening operators and strategies developed in the field could be used for that, we have been able to obtain pretty good experimental results already with rather unsophisticated widening strategies, as we discuss in Section 5.

4 On-the-fly invariant generation

A one-state formula $F[\mathbf{x}]$ is an *invariant for* S if $[\![F]\!]$ includes the set R_S of all reachable states of S. Invariants have many useful applications in static analysis, logic-based model checking, and deductive verification in general. In our abstract domain $\mathbf{E}$ from the previous section, any formula F such that $\mathrm{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(f_{\mathbf{E}}) \sqsubseteq_{\mathbf{E}} [F]$ is an invariant, since $R_S = [\![\mathrm{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(f_{\mathbf{E}})]\!] \subseteq [\![F]\!]$.[7] By the construction of our abstraction in the domain $\mathbf{A}$, any fix point computation for the transformer $g_{\mathbf{A}} : \mathbf{A} \to \mathbf{A}$ starting with the element $I_{\mathbf{A}}$ from Proposition 2 produces a value a such that $\gamma_{\mathbf{F}}(a)$ is an invariant for S.

A distinguishing feature of our approach is that, in practice, we can modify the fix point computation for $g_{\mathbf{A}}$ to generate *intermediate* invariants as it goes and *before* reaching the fix point. We capitalize on the fact that $\gamma_{\mathbf{F}}(a)$ is typically a conjunction of formulas, or *state properties*, $P_1[\mathbf{x}], \ldots, P_m[\mathbf{x}]$. For any intermediate value $a \in \mathbf{A}$ constructed during the fix point computation for $g_{\mathbf{A}}$, if $\gamma_{\mathbf{F}}(a) = P_1 \wedge \cdots \wedge P_m$ we can check whether any of the P_i's is already invariant.

Since the fix point computation using $g_{\mathbf{A}}$ starts with an over-approximation of the initial states, we know that the whole $\gamma_{\mathbf{F}}(a)$ is inductive, and hence invariant, if the satisfiability test on the formula F in Figure 2 fails. However, it is possible to do better by turning that test into one that checks the *k-inductiveness* [27] of the individual P_i's simultaneously. We discuss an efficient mechanism for doing that in previous work [18]. We refer the reader to that work for more details, but the important point here is that,

[7] Of course, obtaining a formula from the equivalence class $\mathrm{lfp}^{\sqsubseteq_{\mathbf{E}}}_{[I]}(f_{\mathbf{E}})$ would be enough for all analysis purposes since that class consists of the strongest invariant for S. However, in general, such formulas may be infinitary or impractical to compute.

given a bound on k, we can identify fairly quickly, for each $i = 0, \ldots, k$, which subsets of $\{P_1, \ldots, P_m\}$ are conjunctively i-inductive.[8] If the whole $\{P_1, \ldots, P_m\}$ is proven k-inductive, which is equivalent to proving that the formula

$$G[\mathbf{x}_0, \ldots, \mathbf{x}_{k+1}] \stackrel{\text{def}}{=} \gamma_{\mathbf{F}}(a)[\mathbf{x}_0] \wedge T[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge \gamma_{\mathbf{F}}(a)[\mathbf{x}_k] \wedge T[\mathbf{x}_k, \mathbf{x}_{k+1}] \wedge \neg\gamma_{\mathbf{F}}(a)[\mathbf{x}_{k+1}]$$

is unsatisfiable, then $g_{\mathbf{A}}$ can return a because in that case it is a fix point. Otherwise, the state $\mathbf{v}_{k+1}$ from a state tuple $(\mathbf{v}_0, \ldots, \mathbf{v}_{k+1})$ that satisfies G can be used to generalize a as done with $\mathbf{v}'$ in Figure 2. In either case, any subset of $\{P_1, \ldots, P_n\}$ that has been proven k-inductive can be output as a set of (intermediate) invariants.

This in effect turns an abstract interpreter for $\mathbf{A}$ using $g_{\mathbf{A}}$ into an on-the-fly invariant generator. The invariants generated in the earlier iterations of the interpreter are usually, but not necessarily, the simplest ones (e.g., interval bounds on a variable, equalities between variables, and so on) and become increasingly more elaborate as the computation proceeds. The main point is that one does not need to wait until the end of a possibly complex fix point computation using a wide set of costly abstractions to obtain potentially useful invariants. Our experimental results confirm this conjecture.

An additional, if secondary, benefit of identifying intermediate invariants is that they can be used to improve the preciseness of later iterations of the very fix point computation that generated them. This can be done by maintaining at all times a conjunction $J[\mathbf{x}]$ of all the intermediate invariants generated until then, and using at each call of $g_{\mathbf{A}}$ the formula $T_J[\mathbf{x}, \mathbf{x}'] \stackrel{\text{def}}{=} T[\mathbf{x}, \mathbf{x}'] \wedge J[\mathbf{x}]$ in place of the original transition relation formula T. Using the strengthened transition formula T_J helps counterbalance the loss of precision caused by widening while maintaining the soundness of $g_{\mathbf{A}}$—since the strengthening discards only states that are definitely unreachable for not satisfying the invariant $J[\mathbf{x}]$.

Application: invariant generation for Lustre programs

This work was originally motivated by the problem of proving invariant properties of Lustre programs. Lustre [15] is a synchronous data-flow specification/programming language with infinite streams of values of three basic types: Booleans, integers, and reals. It is used to model control software in embedded devices. Properties to be proven are typically introduced within Lustre programs as observer Boolean streams so that checking that a property is invariant amounts to checking that its corresponding stream is constantly true. In previous work, we developed a k-induction-based parallel model checker for Lustre programs, called Kind [20], which uses SMT solvers as its main reasoning engine. Kind benefits from the use of auxiliary invariant generators to strengthen its basic k-induction procedure [19]. We implemented the fix point computation method described here as an additional on-line invariant generator for Kind.

Kind works with an idealized version of Lustre with infinite-precision numerical types. Idealized Lustre programs can be readily recast as transition systems in a three-sorted concrete domain with Booleans and (mathematical) integers and reals. Such

[8] For the reader unfamiliar with k-induction, it is enough to know that every k-inductive formula is invariant, and is k'-inductive for every $k' > k$. Also, 0-inductive formulas are inductive in the traditional sense.

```
1 node p_count (a,b,c:bool) returns (x,y:int; obs:bool);
2 var n1, n2:int;
3 let
4   n1 = 10000; n2 = 5000;
5   x = 0 -> if b or c then 0 else if a and (pre x) < n1 then (pre x) + 1 else pre x;
6   y = 0 -> if c then 0 else if a and (pre y) < n2 then (pre y) + 1 else pre y;
7   obs = (x != n1) or (y = n2);
8 tel
```

Fig. 3: Double counter example in Lustre.

systems can be almost directly encoded and reasoned about in a quantifier-free logic of mixed integer and real arithmetic with uninterpreted function symbols. The linear fragment of that logic, which we could call QF_UFLIRA in the nomenclature of SMT-LIB [3], can be efficiently decided by the SMT solvers used by Kind. This means that Lustre programs limited to linear arithmetic are amenable to analysis with our method.

We have built an abstract interpreter, called Kind-AI, for such Lustre programs that computes the abstract transformer automatically as explained earlier, and generates a stream of invariants (for Kind's benefit) during its fix point computation.[9] As abstract domain we use one defined, as usual, as a reduced product of a variety of abstract domains, including relational and non-relational ones. Our implementation of the function $\gamma_{\mathbf{F}}$ converts abstract elements into formulas of QF_UFLIRA as one would expect: an interval $[a;b]$ for a variable x is converted into the formula $a \leq x \wedge x \leq b$; a linear constraint $\Sigma_i\, a_i \cdot x_i \geq c$ is mapped directly to the corresponding formula of QF_UFLIRA. The translation is extended homomorphically to more complex elements. For instance, elements that are the meet of other ones (such as polyhedra, etc.) are converted to the conjunction of the translation of the components.

Kind-AI is written in OCaml and relies on the APRON abstract domain library [17]. It shares with Kind, also written in OCaml, modules to encode Lustre programs as transition systems in the QF_UFLIRA logic, and to interact with an SMT solver. A basic partitioning mechanism allows Kind-AI to express certain conditional properties. Specifically, it is possible to specify any Boolean term or finite range term t from the Lustre program as a *partitioning variable*. Then the premises of the conditional properties are conjunctions of predicates of the form $t = v$, where v is one of the possible values of t. We illustrate the use of Kind-AI here with a typical example: counters, which are used widely within safety mechanisms for critical systems.

Example 1. In the Lustre program shown in Figure 3, two counters x and y are incremented up to their respective maximum value whenever the input value a is **true**; both are reset to 0 when the input c is **true**. The counter x is reset also when the input b is **true**. Suppose we would like to prove that whenever x reaches its maximum value, so does y. This property is expressed by the synchronous observer obs. It is enough to show then that the Boolean stream obs is equal to the constant stream **true**.

With a partitioning using the Boolean terms $x < n_1$ and $y < n_2$, chosen for being if-then-else guards in the program that involve stateful variables, Kind-AI behaves as

[9] Kind-AI and the input problems used in the experiments described in the next section can be found at `http://clc.cs.uiowa.edu/Kind/NFM13`.

follows with respect to the state variable tuple (x, y). Its fix point algorithm finds and injects, in order, into the abstract domain the states $(0, 0)$, $(0, 1)$, $(1, 1)$ and $(2, 2)$. After the injection of $(1, 1)$, the computed abstract element contains the sub-properties $0 \leq x,\ x \leq 1,\ 0 \leq y,\ y \leq 1$ and $x \leq y$. After the injection of $(2, 2)$, Kind-AI identifies three sub-properties as invariants: $0 \leq x$, $0 \leq y$ and $y < n_2 \Rightarrow x \leq y$.[10] Using the same widening heuristics described in the next section, a fix point that also includes the invariants $x \leq 10000$ and $y \leq 5000$ is reached in 3.95 seconds after 31 iterations.

With this program, using k-induction alone Kind is not able to prove in reasonable time the property expressed by obs. However, when run concurrently with Kind-AI, Kind is able to prove the target property as soon as it receives the intermediate invariants $0 \leq x,\ x \leq 10000,\ 0 \leq y,\ y \leq 5000$ and $y < n_2 \Rightarrow x \leq y$. □

5 Experimental evaluation

Our approach relies heavily on widening in practice to ensure convergence. As a consequence, one might wonder about the logical strength of the invariants produced by our invariant generator. To evaluate that we did an initial experimental comparison with a couple of other static analysis tools, ASPIC and SMT-AI, that can generate linear numerical invariants for (finite and) infinite-state systems. The first is a tool combining linear relation analysis with widening and acceleration techniques [11]. The second tool is an abstract interpreter that targets specifically Lustre programs and employs a number of AI techniques to produce program invariants [25].[11]

We looked at the set of infinite-state transition systems collected by Gonnord on the ASPIC website [1]. These are mostly toy numerical systems, specified in the FAST language [9], which however admit interesting conditional and unconditional numerical invariants. FAST expresses transition systems essentially as unbounded counter automata, with a finite control structure and transitions that have linear integer arithmetic guards, and effects described by affine functions. We translated each automaton to a Lustre program by encoding the automaton's states by means of a *mode variable*, a finite range variable with each value representing one of the states.

We ran four different configurations of ASPIC on the FAST systems. We also ran Kind-AI and SMT-AI on the corresponding Lustre programs, with partitioning over the mode variable above and with the *full packs* option, which builds a relational abstraction (using polyhedra and octagons in Kind-AI, and just polyhedra in SMT-AI) on all the stateful variables of the program. In Kind-AI, we used a very simple widening heuristics, which applies widening every two join operations and uses as widening thresholds the numerical constants in the input program. We set an upper bound of 4 for the k-induction loop used in the computation of the abstract transformer $g_{\mathbf{A}}$ described in Section 4. All tests were executed with a 60 second timeout on a Linux machine with a quad-core 2.80 GHz Xeon processor with 12 GB of RAM.

Finally, we compared for each problem the invariants generated by ASPIC and SMT-AI at the end of their analysis with the conjunction of the intermediate invariants

[10] Note that fast pre-analyses used in abstract interpretation tools, such as constant propagation, will not produce implications like the one above.

[11] The "SMT" in the name is just because it works with formulas in the SMT-LIB format [3].

Benchmarks	ASPIC				SMT-AI	Kind-AI	SMT-AI
	Ch79	Ch79V2	Lookahead	Native		runtime	runtime
apache1	= 004	= 004	= 004	= 004	+ 004	005	005
car7	||	−	||	−	||	12,120	083
dummy1	= 003	= 003	−	−	+ 003	005	003
dummy4	+ 014	= 014	= 014	= 014	+ 006	014	005
dummy6	+ 004	+ 028	+ 028	−	+ 002	028	timeout
gb	+ 783	+ 783	= 783	||	+ 009	7,830	026
goubault1b	+ 011	= 011	= 011	= 011	+ 011	026	025
goubault2b	+ 057	+ 072	+ 072	+ 072	+ 057	102	018
hal79a	||	−	−	−	+ 047	1,430	024
hal79b	+ 101	+ 101	−	−	+ 101	1,020	021
simplecar	+ 066	−	= 066	−	+ 005	066	006
sp	||	−	||	||	+ 035	12,300	timeout
subway	||	−	||	||	||	19,130	05,330
swap	||	−	−	−	+ 022	022	006
t4x0	+ 014	+ 014	+ 014	+ 014	+ 014	067	027
train1	||	−	||	||	||	19,040	05,330
wcet1	||	||	||	−	||	5,530	027
wcet2	||	||	||	||	||	38,870	2,270

Fig. 4: Comparison of final invariants computed by Kind-AI vs. those computed by the other tools. The symbol + means Kind-AI's invariant is stronger; – weaker; = equivalent; and || incomparable. All runtimes are in milliseconds.

progressively generated by Kind-AI. Figure 4 summarizes the results of this comparison. The various configuration of ASPIC are explained in [1]. The first three implement earlier methods developed by others [14, 2, 12]; the last one corresponds to ASPIC's own method. The last two columns in the figure show the time SMT-AI and Kind-AI respectively took to compute their fix point. The corresponding runtimes for ASPIC are not reported because they were 3ms in almost all cases, with a maximum of 7ms. The numbers in the ASPIC and SMT-AI columns indicate at what time during Kind-AI's computation the conjunction of its intermediate invariants became equivalent or stronger than the final invariant generated by the other tools.

We stress that Kind-AI was designed to quickly compute auxiliary invariants for Kind, not to produce comprehensive analyses. So it incorporates none of the sophisticated techniques used by ASPIC to increase the precision of its analysis [11]. In spite of that, in many cases it computed stronger or equivalent invariants. This suggests that the sound abstract transformers generated automatically with our method can produce fairly accurate analyses out of the box. The results also confirm that while convergence to a fix point may take considerably longer in Kind-AI than in the other tools, good invariants (i.e., stronger or equivalent to those from the other tools) are produced a lot sooner.

6 Conclusion and further work

The framework we presented offers two main contributions: (*i*) a systematic and automatic generation of abstract transformers based on a combination of logic solvers and abstract domain libraries; (*ii*) the gradual generation of invariants during the computation of post fix points. Our approach is truly automatic whenever the target system can be encoded in a suitable decidable logic and abstract domain elements are representable

in that logic. Such conditions are often easy to satisfy for systems already analyzable with SMT solvers, and for numerous abstract domains. Thanks to continuous advances in SMT, we expect that more and more domains, such as those for finite precision integers and floating point numbers, will be supported by SMT solvers. Our approach will then immediately provide for free abstract interpreters/invariant generators for them. Although our current implementation works with Lustre programs, our general method is language independent. Also, it imposes no restrictions on the abstract domains that can be used as long as, in essence, the domains admit a concretization in a decidable logic with an available solver. Furthermore, our framework facilitates the expression of big-step semantics (on the logical side) and therefore avoids the loss of precision obtained when applying abstract transfer functions at a small-step semantics level.

About the second contribution, to our knowledge, our initial implementation of the framework is the only available tool based on abstract interpretation and Kleene-style fix point computation that provides invariants *before* the post fix point is reached. Even if reduced domains share knowledge about their current state, this information is not a guaranteed fix point and cannot be soundly communicated to other tools. In a multi-analyzer setting, the ability to share invariants before the end of the computation can drastically increase performance. But that sort of intermediate but guaranteed information can be extremely valuable even in standalone use. For example, when statically analyzing a 200k-loc critical embedded software for the absence of run time errors [6], one could start looking at sections of the code that are already proven to be error free while the automatic analysis continues. This contrasts with the current general practice for least-fix point approximations where one gets at most alarms during the computation and has to wait, possibly for hours, for that computation to end before interpreting the results, and realizing perhaps that certain parameters need further tuning.

We have implemented our method and verified the general quality of its generated invariants with a comparative evaluation on some benchmarks admitting interesting numerical invariants. Further work will involve a more extensive experimental evaluation of the method to assess the effects of its generated invariants on the performance of our Kind model checker, which already relies on auxiliary invariants generated by other means. One source of imprecision in our method, leading to weaker invariants, is the generalization of the current abstract value to include successor states that may in fact be unreachable. Additional work will focus on developing enhancements for mitigating this problem.

References

1. Aspic website. `http://laure.gonnord.org/pro/aspic/benchmarks.html`.
2. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *SAS*, volume 2694 of *LNCS*, pages 337–354, 2003.
3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *ESOP*, volume 3444 of *LNCS*, pages 21–30, 2005.
7. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FOSSACS*, volume 6604 of *LNCS*, pages 456–472, 2011.
8. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.
9. Fast website. `http://www.lsv.ens-cachan.fr/fast/`.
10. T. M. Gawlitza and D. Monniaux. Improving strategies via SMT solving. In *ESOP*, volume 6602 of *LNCS*, pages 236–255, 2011.
11. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS*, volume 4134 of *LNCS*, pages 144–160, 2006.
12. D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, volume 4144 of *LNCS*, pages 452–466, 2006.
13. A. Gurfinkel and S. Chaki. Combining predicate and numeric abstraction for software model checking. *STTT*, 12(6):409–427, 2010.
14. N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, University of Grenoble, 1979.
15. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
16. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *SAS*, volume 1694 of *LNCS*, pages 39–50, 1999.
17. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667, 2009.
18. T. Kahsai, P.-L. Garoche, C. Tinelli, and M. Whalen. Incremental verification with mode variable invariants in state machines. In *NFM*, volume 7226 of *LNCS*, pages 388–402, 2012.
19. T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NFM*, volume 6617 of *LNCS*, pages 192–207, 2011.
20. T. Kahsai and C. Tinelli. PKIND: a parallel *k*-induction based model checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.
21. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213, 2010.
22. D. Monniaux. Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151. ACM, 2009.
23. D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS*, volume 6887 of *LNCS*, pages 369–385, 2011.
24. T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266, 2004.
25. P. Roux, R. Delmas, and P.-L. Garoche. SMT-AI: an abstract interpreter as oracle for k-induction. *Electr. Notes Theor. Comput. Sci.*, 267(2):55–68, 2010.
26. P. Schrammel and B. Jeannet. Extending abstract acceleration methods to data-flow programs with numerical inputs. *Electr. Notes Theor. Comput. Sci.*, 267(1):101–114, 2010.
27. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 127–144, 2000.
28. A. V. Thakur, M. Elder, and T. W. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, volume 7460 of *LNCS*, pages 111–128, 2012.
29. A. V. Thakur and T. W. Reps. A method for symbolic computation of abstract operations. In *CAV*, volume 7358 of *LNCS*, pages 174–192, 2012.
30. A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In *CADE*, volume 4603 of *LNCS*, pages 147–166, 2007.
31. G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, volume 2988 of *LNCS*, pages 530–545, 2004.