Static Analysis, Abstract Interpretation and Verification in (Constraint Logic) Programming

Giorgio Delzanno[1], Roberto Giacobazzi[2], and Francesco Ranzato[3]

[1] Università di Genova, Italy, giorgio@disi.unige.it
[2] Università di Verona, Italy, roberto.giacobazzi@univr.it
[3] Università di Padova, Italy, francesco.ranzato@unipd.it

Introduction

Logic programming has served as a unique training ground for static analysis, abstract interpretation and verification. Operational and denotational semantics of logic programs feature simple and clean inductive definitions that made it possible to apply a variety of known analysis and verification techniques and tools and to define new ones tailored to solve specific problems arisen in logic programming (e.g. variable aliasing and unification). We survey here some general notions and methods — in particular abstract interpretation and model checking — for analysing and verifying programs and systems, especially focused to (constraint) logic programs.

In Section 1 we first review the principles of the abstract interpretation approach, in particular methodologies for designing abstract domains through systematic techniques such as abstract domain refinement and simplification. We then show how these methods have been applied in the systematic design of analyses and semantics in the context of logic programming.

In Section 2 we recall the main concepts underlying model checking. In model checking, the behavior of a concurrent system is described by a finite graph (a Kripke model) that describes the set of all reachable states. In this setting, temporal formulae can be used to naturally specify functional properties of the system (e.g. safety and absence of starvation). The model checking problem consists in checking the temporal specification against the model of the system. For specifications given in Computation Tree Logic (CTL), the algorithm for deciding the model checking problem is based on a fixpoint semantics of the temporal connectives. We exploit here this connection to establish a link between CTL model checking and the fixpoint semantics of logic programs. We then discuss implications of this link with a particular focus on the utilization of evaluation strategies used for logic programming as a tool for model checking of infinite-state concurrent systems.

In Section 3 we focus on abstract interpretation-based model checking. In abstract model checking, the verification of a temporal specification is performed in an abstract model that can be designed as an abstract interpretation of the concrete system. In particular, we concentrate on strong preservation properties of abstract models, namely on the equivalence of verifying temporal specifications in abstract and concrete models. Strong preservation is highly desirable since it allows us to draw consequences on the concrete model from negative answers on the abstract model. We survey how abstract

interpretation allows to cast strong preservation as a completeness property of abstract models and consequently how this provides systematic methods to design strongly preserving abstract models through abstract domain refinements.

Finally, in Section 4 we discuss how methods used for evaluation and analysis of logic programs can be used to extend verification methods based on abstract model checking, e.g., to the case of infinite-state systems.

1 Semantics, Static Analysis and Abstract Interpretation

1.1 Abstract Interpretation Basics

One fundamental feature of abstract interpretation is that most properties in approximating semantics, like precision, completeness, and compositionality, which may involve complex operators, fixpoints etc., all depend upon the notion of *abstraction*, which is precisely and uniquely determined by the chosen domain of properties [16]. Central in the design of abstract interpretations is therefore the notion of *domain*. This is the case for instance in program analysis, in type inference and in comparative semantics, where the various abstract (approximate) semantics all correspond to suitable abstractions, namely domains.

In the following, $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a generic complete lattice C, with ordering $\leq$, lub $\vee$, glb $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. The downward closure of a subset $S \subseteq C$ is defined as $\downarrow S \triangleq \{x \in C \mid \exists y \in S.\ x \leq y\}$, where $\downarrow x$ is a shorthand for $\downarrow \{x\}$. The upward closure $\uparrow$ is dually defined. The notation $C \cong D$ denotes that C and D are isomorphic, possibly ordered, structures. Recall that a function $f : C \to D$ is (Scott-)continuous if f preserves lub's of (nonempty) chains iff f preserves lub's of directed subsets. In what follows, we consider abstract interpretation based on Galois connections or, equivalently, closure operators [15, 16]. A pair of functions $f : A \to B$ and $g : B \to A$ between posets forms an adjunction, or Galois connection (GC for short), denoted by (A, f, B, g), if

$$\forall x \in A. \forall y \in B.\ f(x) \leq_B y \Leftrightarrow x \leq_A g(y).$$

f (resp. g) is called the left- (right-) adjoint to g (f) and it is an additive (co-additive) function, i.e., f preserves lub's (glb's) of all subsets of A (empty set included). Additive and co-additive functions f admit, respectively, right f^+ and left f^- adjoint as follows: $f^+ \triangleq \lambda x. \vee \{y \mid f(y) \leq x\}$ and $f^- \triangleq \lambda x. \wedge \{y \mid x \leq f(y)\}$. Let us also recall that $(f^+)^- = (f^-)^+ = f$. In GC-based abstract interpretation the concrete C and abstract A domains are often assumed to be complete lattices and are related by abstraction $\alpha : C \to A$ and concretization $\gamma : A \to C$ maps forming a GC (C, α, A, γ). If in addition $\forall a \in A.\ \alpha(\gamma(a)) = a$, then (C, α, A, γ) is called a Galois insertion (GI). When (C, α, A, γ) is a GI each value of the abstract domain A is useful in representing C, namely all the elements of A represent distinct members of C, being γ 1-1. Any GC may be lifted to a GI by identifying in an equivalence class those values of the abstract domain with the same concretization. This process is known as reduction of the abstract domain. An (upper) closure operator on a poset C is a map $\rho : C \to C$ which is monotone, idempotent, and extensive ($\forall x \in C.\ x \leq \rho(x)$). The set of all closure

operators on C is denoted by $uco(C)$. Each closure operator ρ is uniquely determined by its image $\rho(C)$ as follows: $\rho(x) \triangleq \wedge\{y \in \rho(C) \mid x \leq y\}$. A fundamental property of closure operators is that if C is a complete lattice then both $\langle uco(C), \sqsubseteq\rangle$, where $\sqcup$ is the pointwise ordering, and $\langle \rho(C), \leq_C\rangle$ are complete lattices. It is well known since [16] that abstract domains can be equivalently specified either as Galois insertions or as closure operators on the concrete domain. In particular, a subset $X \subseteq C$ is the image of a closure ρ on C iff X is a Moore-family of C, i.e., $X = \mathcal{M}(X) \triangleq \{\wedge S \in C \mid S \subseteq X\}$ (where $\wedge\varnothing = \top \in \mathcal{M}(X)$) iff X is isomorphic to an abstract domain A in a GI (C, α, A, γ). For any subset $X \subseteq C$, $\mathcal{M}(X)$ is called the Moore-closure of X in C, i.e., $\mathcal{M}(X)$ is the least (w.r.t. set-inclusion) subset of C which contains X and it is a Moore-family of C. $\langle uco(C), \sqsubseteq\rangle$ is isomorphic to the so-called lattice $\langle \mathrm{Abs}(C), \sqsubseteq\rangle$ of abstract interpretations of C [16]. Hence, given any two abstractions $A, B \in \mathrm{Abs}(C)$, A is more precise (or conrete) than B, denoted by $A \sqsubseteq B$, when $B \subseteq A$ as Moore families of C. In the following, it is particularly convenient to identify an abstract domain $A \in \mathrm{Abs}(C)$ as (image of) a closure operator on C, which, as a function, is denoted by ρ_A.

1.2 Backward and Forward Completeness

Soundness of an abstraction can be specified in two equivalent ways [15]. Let C be a concrete domain, (C, α, A, γ) a Galois insertion, $f : C \to C$ a concrete semantic operation and $f^\sharp : A \to A$ a corresponding abstract operation. Then, (C, α, A, γ) and $f^\sharp$ give rise to a sound abstraction when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$, or equivalently (by adjunction) when $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$. While the above two definitions of soundness are equivalent, it turns out that they are not equivalent when equality is required and they encode two different forms of completeness: in the first case, $\alpha \circ f = f^\sharp \circ \alpha$ is called backward ($\mathcal{B}$-) completeness while $f \circ \gamma = \gamma \circ f^\sharp$ is called forward ($\mathcal{F}$-) completeness — the reason for these names will be clear later in the paper. $\mathcal{B}$-completeness (see [44]) corresponds to ask that the abstract function $f^\sharp$ perfectly mimics the concrete function f when the latter is approximated in A, viz. both functions are compared in the abstract domain A. On the other hand, $\mathcal{F}$-completeness (see [37]) corresponds to ask that $f^\sharp$ perfectly mimics the function f when applied to the same abstract value, viz. they are both compared in the concrete domain C.

Recall that the best correct approximation of f on the abstract domain A is defined to be the abstract function $\alpha \circ f \circ \gamma$. It turns out (this is a simple extension of a characterization in [44]) that, given an abstract domain A, there exists an either $\mathcal{B}$- or $\mathcal{F}$-complete abstract function $f^\sharp$ defined on A iff the best correct approximation of f on A is, respectively, either $\mathcal{B}$- or $\mathcal{F}$-complete. This means that both $\mathcal{B}$- and $\mathcal{F}$-completeness are properties of abstract domains, namely a property of the GI (C, α, A, γ). Therefore, one may define $\mathcal{B}$- and $\mathcal{F}$-completeness as follows: an abstract domain $A \in \mathrm{Abs}(C)$ is $\mathcal{B}$-($\mathcal{F}$-) complete for a semantic function f if $\rho_A \circ f = \rho_A \circ f \circ \rho_A$ ($f \circ \rho_A = \rho_A \circ f \circ \rho_A$).

While $\mathcal{B}$-completeness is well known in abstract interpretation and corresponds to the standard notion of completeness [44, 59], the notion of forward completeness is less known. $\mathcal{B}$-completeness for a domain A means that the expressive power of A is such that no loss of precision is accumulated in A by abstracting in A itself the arguments of a semantic function f. Conversely, $\mathcal{F}$-completeness means that no loss of precision is accumulated by approximating in A the result of the function f when computed on

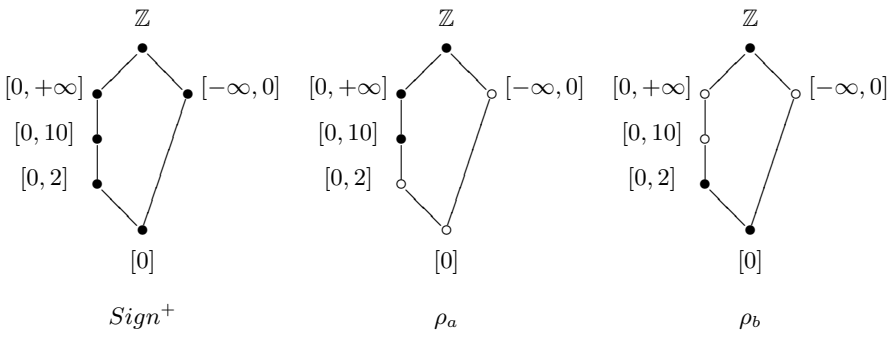

Fig. 1. The abstract domain $Sign^+$ and two abstractions

abstract values in A. This justifies the choice of the backward and forward terminology above. We denote by, respectively, $\mathcal{F}(C, f)$ and $\mathcal{B}(C, f)$ the set of $\mathcal{F}$- and $\mathcal{B}$- complete abstractions of C for f. It is worth noting that in general $\mathcal{F}(C, f) \not\subseteq \mathcal{B}(C, f)$ and $\mathcal{F}(C, f) \not\subseteq \mathcal{B}(C, f)$, namely $\mathcal{B}$- and $\mathcal{F}$-completeness are incomparable notions.

Example 1. Let $Sign^+$ be the simple abstraction of $\langle \wp(\mathbb{Z}), \subseteq \rangle$ for analysing integer variables depicted in Fig. 1. Consider the pointwise square operation $sq : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ defined as follows: $sq(X) \triangleq \{x^2 \mid x \in X\}$. Let $\rho \in uco(\wp(\mathbb{Z}))$ be the closure operator associated with $Sign^+$, i.e. $\rho = \gamma_{Sign^+} \circ \alpha_{Sign^+}$, where the abstraction and concretization maps are the obvious ones. The best correct approximation of sq in $Sign^+$ is $sq^\sharp : Sign^+ \to Sign^+$ defined as $sq^\sharp(X) \triangleq \rho(sq(X))$, with $X \in Sign^+$. It is easy to note that the closure operators $\rho_a \triangleq \{\mathbb{Z}, [0,+\infty], [0,10]\}$ and $\rho_b \triangleq \{\mathbb{Z}, [0,2], [0]\}$, defined by their images — the images of ρ_a and ρ_b are depicted as bullets in Fig. 1 — are such that:

- $\rho_a \in \mathcal{F}(Sign^+, sq^\sharp)$ but $\rho_a \notin \mathcal{B}(Sign^+, sq^\sharp)$: for example, $\rho_a(sq^\sharp(\rho_a([0]))) = [0,+\infty]$ while $\rho_a(sq^\sharp([0])) = [0,10]$;
- $\rho_b \in \mathcal{B}(Sign^+, sq^\sharp)$ but $\rho_b \notin \mathcal{F}(Sign^+, sq^\sharp)$: for example, $\rho_b(sq^\sharp(\rho_b([0,2]))) = \mathbb{Z}$ while $sq^\sharp(\rho_b([0,2])) = [0,10]$. □

One key result in [44] provides a constructive characterization of the structure of abstract domains that are $\mathcal{B}$-complete for continuous functions. Given a function $f : C \to C$ and $S \subseteq C$, $f^{-1}(S)$ denotes the inverse image of f in S, i.e., $\{x \in C \mid f(x) \in S\}$. Then, [44] shows that

$$\rho \in uco(C) \text{ is } \mathcal{B}\text{-complete for } f \Leftrightarrow \bigcup_{y \in \rho(C)} \max(f^{-1}(\downarrow y)) \subseteq \rho(C) \qquad (*)$$

Let us consider Example 1. It is easy to see that ρ_a is not $\mathcal{B}$-complete because ρ_a does not include the maximal inverse image of $sq^\sharp$ of the subset $\downarrow [0,10]$, namely $\max({sq^\sharp}^{-1}(\downarrow [0,10])) = \{[0,2]\}$.

An analogous (and trivial to prove) result can be stated for $\mathcal{F}$-completeness. In this case, $\mathcal{F}$-complete domains can be characterized for merely monotone operations as

follows:

$$\rho \in uco(C) \text{ is } \mathcal{F}\text{-complete for } f \Leftrightarrow f(\rho(C)) \subseteq \rho(C) \qquad (**)$$

Thus, while $\mathcal{B}$-complete domains ρ are closed under (maximal) inverse images of the function f on $\rho(C)$, $\mathcal{F}$-complete domains ρ are closed under direct images of f on $\rho(C)$. It is easy to see in Example 1 that ρ_b is not $\mathcal{F}$-complete because ρ_b does not include the direct image of $sq^\sharp$, for instance the value $[0, 10] = sq^\sharp([0, 2])$. Characterizations $(*)$ and $(**)$ together establish a tight relationship between $\mathcal{B}$- and $\mathcal{F}$-completeness, which can be specified as an adjunction when the concrete function admits a right adjoint. In fact, it turns out that if $f : C \to C$ is an additive function (and therefore admits right adjoint f^+) then

$$\mathcal{B}(\wp(S), f) = \mathcal{F}(\wp(S), f^+). \qquad (\ddagger)$$

Moreover, it is always possible, by relying on $(*)$ and $(**)$, to associate with each continuous semantic function $f : C \to C$ a corresponding domain refinement that transforms any abstract domain A into the closest (most abstract) $\mathcal{B}$-/$\mathcal{F}$-complete domain for f which includes (i.e., is more precise than) A. This provides the notions of $\mathcal{B}$- and $\mathcal{F}$-complete shell [44]. The domain transformers $\mathcal{R}^{\mathcal{B}}_f : uco(C) \to uco(C)$ and $\mathcal{R}^{\mathcal{F}}_f : uco(C) \to uco(C)$ are defined as follows:

- $\mathcal{R}^{\mathcal{B}}_f \triangleq \lambda X \in uco(C).\ \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y)))$;
- $\mathcal{R}^{\mathcal{F}}_f \triangleq \lambda X \in uco(C).\ \mathcal{M}(f(X))$.

It is immediate to note that both $\mathcal{R}^{\mathcal{B}}_f$ and $\mathcal{R}^{\mathcal{F}}_f$ are monotone operators on $uco(C)$. The following equivalence, which follows from $(*)$ and $(**)$, characterizes in a unique domain-equational form the $\mathcal{B}$-/$\mathcal{F}$-complete shell of abstract domains for a continuous function $f : C \to C$. Let $A \in uco(C)$ and $\ell \in \{\mathcal{B}, \mathcal{F}\}$:

$$X \sqsubseteq A \text{ and } X \text{ is } \ell\text{-complete for } f \ \Leftrightarrow\ X = A \sqcap \mathcal{R}^{\ell}_f(X).$$

Therefore, the most abstract domain that includes A and is ℓ-complete for f is

$$\ell\text{-}\operatorname{Shell}_f(A) \triangleq \operatorname{gfp}(\lambda X.A \sqcap \mathcal{R}^{\ell}_f(X)).$$

This domain is called the ℓ-complete shell of A with respect to f.

1.3 Abstract Domain Refinement and Simplification

In recent years, systematic design methods of program analysis frameworks attracted a growing interest. This is mainly justified by the fact that the most successful static analyzers are parametric with respect to the property of interest [20] and therefore allow to easily handle a variety of possible analyses. Moreover, automatic methods for tuning static analyses in accuracy and cost are needed in order either to avoid reimplementation when these analyses are modified or to minimize false alarms. Similar constructions are also used in designing semantics by abstract interpretation (e.g., Hoare logic as tensor product [14] and compositional semantics as reduced power [33, 42]) and in

type inference (e.g., polymorphism as disjunctive completion [13, 51]). Formal methods that compare/transform abstract interpretations are therefore inherently based on corresponding methods to compare/transform abstract domains. A domain, at any level of abstraction, is a set of mathematical objects which represent the properties of interest about a computational system and that are partially ordered with respect to their relative degree of precision. In program analysis, for instance, the design of a static analyzer basically corresponds to study a particular abstract domain, while modifying domains corresponds to modify analyses. As shown for instance in [70] for a reconstruction of groundness analysis in logic programming, the design of a complex abstract domain is generally the result of a number of steps which can be in some cases made systematic by applying suitable domain transformers to simpler domains for the property of interest.

The main idea behind domain transformers in abstract interpretation consists in designing abstract domains systematically from the specification of some simpler domains of basic properties of interest and then solving a recursive domain equation in order to achieve completeness with respect to some target precision level. This game can be played for most of the existing abstract domain transformers, by viewing them as instances of completeness refinements: (1) in program analysis, where a given simple (and imprecise) analysis is refined until completeness is reached by avoiding specific families of false alarms, and (2) in program semantics where a given observation is refined towards completeness in order to attain compositionality, condensation properties, etc.

The foundations of a theory of abstract domain transformers were layed by Cousot and Cousot [16] in 1979. In that seminal work the authors introduced the main structure of abstract domains enjoying Galois connections and some fundamental operators for systematically compose domains in order to achieve attribute independent and relational analyses (respectively, the reduced product and reduced power operations). Since then, a number of papers put forward novel domain transformers and studied the impact of these operations in designing abstract interpreters for specific program analysis and languages. These include Cousot and Cousot's reduced product, disjunctive completion and reduced cardinal power [16–18]; Nielson's tensor product [60]; Giacobazzi et al.'s dependencies, dual-Moore-set completion, complete kernels and shells, Heyting completion, and least disjunctive basis [40, 44, 46]; Cortesi et al.'s open product, pattern completion, and complementation [11]. The notions of domain refinement and domain simplification, introduced in [27, 39], provided the very first generalization of these ideas. Intuitively, a refinement is any domain operator that performs an action of refinement with respect to the standard precision ordering $\sqsubseteq$, i.e., that adds information to domains; on the other hand, simplificators and compressors perform the dual action of "taking out" information from domains. Still these operators represent a basis for any design of abstractions.

Many domain refinements can be specified as $\mathcal{F}$-complete refinements with respect to a given semantic operation [35]. Intuitively, a domain refinement can be viewed as adding the functionalities of a given semantic operation of interest, that is, the direct image of a semantic function. As a result of the above properties of complete abstractions, this corresponds to say that a domain refinement can be specified as (greatest) solution of a $\mathcal{F}$-completeness equation. As recalled above in $(\ddagger)$, whenever the seman-

tic operation is additive, such a characterization can be put in an equivalent formulation in terms of $\mathcal{B}$-completeness.

Clearly, the construction of domains by iterative refinement (e.g., by solving a recursive domain equation) may lead to excessively complex domains for practical applications, as well as it may be interesting to isolate inner structures inside complex domains that model precisely some basic properties around which complex abstract domains are built. As observed in [39], it is possible to define a dual theory of domain simplificators and compressors, which shares with the above theory of domain refinements precisely the same, but dual, ideas and constructions. The common aspects of simplificators and compressors is that they both reduce precision in domains. A typical pattern for domain simplificators is the operation that transforms a given domain A into the most concrete (when it exists) among the abstractions of A which is complete for a given function. Like refinements, also simplificators and compressors have a constructive definition as (greatest) solutions of (systems of) recursive domain equations [44]. The main difference between simplificators and compressors can be grasped by viewing how they react when composed with the corresponding refinements, when they exist. Assume that an idempotent refinement $\mathcal{R}$ is given. $\mathcal{R}$ admits a simplificator $\mathcal{S}$ when, for any abstraction X, $\mathcal{R}(\mathcal{S}(X)) = \mathcal{S}(X)$ and $\mathcal{S}(\mathcal{R}(X)) = \mathcal{R}(X)$. This holds when both $\mathcal{R}$ and $\mathcal{S}$ transform domains to meet a given common property, like, for instance in the above case, completeness. A relevant example of domain refinement which has a corresponding simplificator is in fact the complete shell refinement in [44]. The complete shell refinement, given a domain A, returns the most abstract domain which includes A and is complete for some given semantic operation f; the corresponding simplificator, called complete core, returns the most concrete domain which is contained in A and is complete for f. Compressors, instead, act like "zip" runs on files. If $\mathcal{R}$ is a given domain refinement, $\mathcal{C}$ is a compressor for $\mathcal{R}$ if, for any abstraction X, $\mathcal{R}(X) = \mathcal{R}(\mathcal{C}(X))$ and $\mathcal{C}(\mathcal{R}(X)) = \mathcal{C}(X)$, namely when $\mathcal{C}(X)$ is the most abstract domain B such that $\mathcal{R}(B) = \mathcal{R}(X)$, and this basically holds when the whole refined domain $\mathcal{R}(X)$ can be fully reconstructed by refinement from its so-called basis $B = \mathcal{C}(X)$. A domain theoretic definition of abstract domain compressors has been introduced in [41]. Examples of domain compressors include complementation [11, 28], which is the compressor associated with reduced product, and least disjunctive basis [40], which is associated with the disjunctive completion refinement. Clearly, not all refinements admit a corresponding simplificator or compressor. Moreover, as suggested by the above definitions, it is possible to relate refinements and simplificators/compressors by means adjunctions [35, 39].

1.4 How to Cook an Abstract Domain or Semantics

The above methods can be used as a recipe for "cooking" an abstract domain/semantics for specific applications.

1. Specify a concrete semantics for the considered programming language, with a (possibly many sorted algebra as) concrete domain $\mathcal{C} = \langle C, op_1, \ldots, op_n \rangle$;
2. Identify, as a subset of the lattice of abstract interpretations, some basic semantics properties $\pi \subseteq \mathrm{Abs}(C)$ that are to be preserved by the abstraction process;

3. Design a suitable refinement $\mathcal{R}_\pi$ which adds to domains some functionalities of the concrete algebra $\mathcal{C}$, in such a way that $\mathcal{R}_\pi(X) = X \Rightarrow X \in \pi$;
4. Define an adequate abstract domain A that encodes the basic properties of interest (e.g. the basic properties to analyze) concerning concrete data objects;
5. Solve the (system) of recursive domain equations $X = A \sqcap \mathcal{R}_\pi(X)$.

Step (1) is common to any abstract interpretation, and corresponds to the design of a suitable base (typically collecting) semantics. Step (2) is instead a meta-level operation: The designer has to identify the common structure of any domain which shares a given semantic property that has to be preserved in the abstraction process. This may include completeness, compositionality, and any combination of semantic properties of interest for the specific application. A taxonomy of basic observable properties of semantics is essential in order to solve this problem, see e.g. [21] for a recent account on the logic programming case. Step (3) is strongly related to step (2) and is based on the theory of domain refinements described above [39]. Step (4) strongly needs a creative contribution of the designer, which has to guess a minimal domain of basic properties of interest for concrete data objects. Compressors may provide here a tool for simplifying and adapting the solutions envisaged at design time. Steps (5) is standard. Most of these steps, in particular (3) and (5), are systematic and, in most cases, constructive.

1.5 Applications in Logic Programming

Logic programming has been an ideal programming setting where the above ideas have found straight application. This because of the clean nature of the declarative semantics of a (constraint) logic program, which consists of a simple fixpoint solution of a recursive equation on predicates, where ground predicates provide the so called model-based semantics and possibly nonground predicates provide the so called computed-answer substitution semantics, also called s-semantics [26]. This motivates the use of logic programming as a natural and intelligible environment where abstract domain transformers can be tested and applied for a very first practical use, and characterized the research in abstract interpretation in the years across Y2000 mainly in Padova, Parma, Pisa and Verona. Of course, all the above definitions and notions hold on generic complete lattices and semantic structures, fulfilling the language independence feature of abstract interpretation. Here, we list some results in semantics and static program analysis obtained by applying the above mentioned domain transformers. These results are characterized by a scattered coverage of known and new properties of semantics of logic programming, all having a distinctive nature of being systematically derived by means of abstract domain transformations. The result was a puzzle of methods and techniques for handling semantics and analyses with the ambition of fully developing Strachey's programme of *"understanding of the mathematical ideas of programming languages and combine them with other principles of common sense as correctives of exaggeration, allowing the individual reader to draw as moderate conclusions as she/he will"* [73].

Analysis. The abstract domain for relational groundness analysis *Pos* has been reconstructed as solution of a completeness problem, i.e., as greatest (w.r.t. $\sqsubseteq$) solution of

the simple recursive abstract domain equation $X = \mathcal{G} \sqcap (X \to X)$ over the concrete domain of downward-closed sets of idempotent substitutions with respect to variable instantiation, where $\to$ is the Heyting completion of an abstract domain [46] and $\mathcal{G}$ is the basic domain for groundness analysis, specifying whether a variable is ground or not [70]. Disjunctive completion and bases for groundness analysis have been studied in [40].

The phenomenon of so-called condensation in logic program analysis has been fully modeled as a completeness property of the underlying abstract domains in [45]. A static analysis is condensing if (bottom-up) goal-independent and (top-down) goal-dependent analyses agree, i.e., whenever it is possible to reconstruct the analysis of a given goal from the result of a goal-independent analysis without loss of precision. In this case, a condensing domain can always be systematically derived from a possibly noncondensing one A by solving the recursive domain equation $X = A \sqcap X \sqcap (X \stackrel{\wedge}{\multimap} X)$ on the concrete quantale of sets of idempotent substitutions, where the conjunction $\wedge$ in the quantale of idempotent substitutions is most general unification and where $\stackrel{\wedge}{\multimap}$ is the linear refinement with respect to $\wedge$ [45]. Condensing domains for freeness, independence, type representations, pair-independence, non-pair-sharing, and information-flow analysis have all been derived in this way in [45, 55–57, 72]. A condensing domain for sharing analysis, i.e., a solution to the equation $X = \mathit{Sh} \sqcap X \sqcap (X \stackrel{\wedge}{\multimap} X)$, with *Sh* being the domain for set-sharing, is still unknown. Completeness has been also used in combination with complementation to prove that set-sharing is redundant for pair-sharing [3].

Semantics. Semantics can be composed and complemented as easy as abstract domains. Applications in logic programming have shown that the semantics $\mathcal{S} \boxminus \mathcal{C}$, obtained by complementing [11] the Clark semantics of correct answer substitutions $\mathcal{C}$ with respect to the more concrete semantics of computed answer substitution $\mathcal{S}$, corresponds precisely to the fully abstract semantics for partial computed answer substitutions [38]. Similar characterizations have been obtained by domain complementation of Clark vs. Herbrand model-based semantics and call vs. success pattern semantics [38]. By considering linear refinement, the OR-compositional semantics of logic programs can be systematically derived as least solution of the recursive domain equation $X = \mathcal{S} \sqcap X \sqcap (X \stackrel{\frown}{\multimap} X)$ over the concrete quantale of SLD-traces of atoms where conjunction is trace concatenation $\frown$ [42] and $\stackrel{\frown}{\multimap}$ is the linear refinement w.r.t. $\frown$. A more general construction for arbitrary compositional semantics on traces can be found in [34].

2 Temporal Logic and Model Checking

2.1 Basics of Model Checking

Automated verification methods are based on formal specification languages for modeling system and qualitative properties and on automated deduction engines and theorem proving for checking the properties against a model. Model checking [10] is probably the most widely used method for verification of temporal properties against finite-state

models. The model checking problem, written $M \models \varphi$, consists in checking whether a given property specification φ is satisfied by a model M that describes the behavior of the system. In this setting, M is formally defined as a Kripke model, i.e., a finite graph in which nodes are labeled by propositions (the basic properties that hold in a given state) and edges represent transitions between states (transition relation). The property φ is specified by means of a temporal logic formula, a formalism that can be used to reason on the transitive closure of the state transition relation. There exist several types of temporal logic specification languages. Two well known examples are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). In the following, we briefly recall the basics of CTL model checking.

CTL can be used to reason about branching time properties of concurrent systems. A CTL model is a tuple $M = \langle \mathrm{States}, \rightarrow, \ell \rangle$ such that States is a set of states, $\rightarrow \subseteq \mathrm{States} \times \mathrm{States}$ is a (typically total) transition relation and $\ell : \mathrm{States} \rightarrow \wp(\mathrm{Atoms})$ is a labeling function that defines the set of atomic predicates, taken from a finite set Atoms, that holds at each state. When a labeling function is omitted, we assume that $\ell(s) = \{s\}$ (i.e., states are used as predicates).

CTL formulae extend propositional logic with temporal formulae of the form $Q_P Q_T$, where Q_P is a path quantifier and Q_T is a temporal quantifier. The path quantifier can be either A (for all paths) or E (there exists a path). The temporal quantifier can be either X (next state), F (eventually), G (always), or U (until). For instance, the formula $\mathrm{EX}\varphi$ holds in the current state if there exists a successor in which φ holds, $\mathrm{EF}\varphi$ holds in the current state if there exists a path in which φ eventually holds, and $\mathrm{AG}\varphi$ holds in the current state if in all paths φ always holds. To formally define the semantics of CTL formulae, we define a path σ in M as an infinite sequence of states $s_0 s_1 \ldots s_i \ldots$ such that $s_k \rightarrow s_{k+1}$ for $k \geq 0$ and we use $\sigma[i]$ to denote the i-th state in σ. Furthermore, we use $P_M(s)$ to define the set of paths σ in M such that $\sigma[0] = s$. The satisfiability relation $M, s \models \varphi$ is defined then as follows:

- $M, s \models p$ iff $p \in \ell(s)$
- $M, s \models \neg\phi$ iff $s \not\models \phi$
- $M, s \models \varphi \vee \psi$ iff $s \models \varphi$ or $s \models \psi$
- $M, s \models \mathrm{EX}\varphi$ iff $\exists \sigma \in P_M(s). \sigma[1] \models \varphi$
- $M, s \models \mathrm{E}(\varphi \,\mathrm{U}\, \psi)$ iff $\exists \sigma \in P_M(s)\ \exists j \geq 0.\ \sigma[j] \models \psi \wedge (\forall k \in [0, j).\ \sigma[k] \models \varphi)$
- $M, s \models \mathrm{EF}\varphi$ iff $\exists \sigma \in P_M(s)\ \exists j \geq 0. \sigma[j] \models \varphi$
- $M, s \models \mathrm{EG}\varphi$ iff $\exists \sigma \in P_M(s)\ \forall j \geq 0. \sigma[j] \models \varphi$

The semantics of the other logical/temporal operators is derived by exploiting semantic equivalences like $\neg EF\varphi \equiv AG\neg\varphi$. Given a CTL model M, an initial state s_0, and a CTL formula φ, the CTL model checking problem consists in checking whether $M, s_0 \models \varphi$ holds or not.

2.2 Model Checking Algorithm

The model checking decision procedure is based on a fixpoint characterization of the semantics of CTL formulae. Given a formula φ we define its denotation as the set of states that satisfies it, i.e.,

$$[\![\varphi]\!] \triangleq \{s \in \mathrm{States} \mid M, s \models \varphi\}.$$

The set of CTL formulae ordered with respect to the inclusion of their denotations forms a complete lattice. The bottom element is $false$ (any unsatisfiable formula), the top element is $true$ (any tautology), and $\wedge$ and $\vee$ correspond to the greatest lower bound and the least upper bound operations, respectively. Temporal connectives can be viewed as transformers of sets of states (i.e., of denotations). To clarify this point, let us recall that temporal connectives as e.g. EF satisfy expansion axioms like

$$\mathrm{EF}\varphi \equiv \varphi \vee \mathrm{EX}\,\mathrm{EF}\varphi.$$

Lifting this axiom to the denotation level we obtain the fixpoint equation

$$Z = h(Z)$$

where $h : \wp(\mathrm{States}) \rightarrow \wp(\mathrm{States})$ is defined as

$$h \triangleq \lambda Z.[\![\varphi]\!] \cup Pre(Z)$$

where $Pre(Z)$ is the set of predecessor states of Z, i.e.,

$$Pre(Z) \triangleq \{s \in \mathrm{States} \mid \exists s' \in Z.s \rightarrow s'\}.$$

The denotation of the formula $\mathrm{EF}\varphi$ is the *least fixpoint* of the operator h, which is monotonic over the complete lattice $\langle \wp(\mathrm{States}), \subseteq, \cup, \cap, \mathrm{States}, \varnothing \rangle$. By applying Knaster-Tarski fixpoint theorem, the least fixpoint of h is the union $\bigcup_{i \geq 0} I_i$ of the sets $I_0, \ldots, I_i, \ldots$ inductively defined as $I_0 = \varnothing$ and $I_{i+1} = h(I_i)$ for $i \geq 0$. This computation corresponds to a backward visit of the graph that defines the state transition relation starting from the set of states that satisfy φ. Since the model has finitely many states this backward analysis is always guaranteed to terminate and requires a number of steps that is linear in the size of the model (in the worst case one state is added in each computation of Pre).

A similar reasoning can be applied to the other CTL connectives. The denotation of formulae that quantify over all states along a path, like AG and EG, can be computed as greatest fixpoints of their corresponding transformers, whereas the denotation of temporal formulae like AF and EF can be computed as least fixpoints. The model checking algorithm is defined then by induction on the structure of the input formula φ and computes its denotations bottom-up starting from the denotations of its subformulae. For instance, given the formula $\mathrm{AG}((\mathrm{EF}\ p) \wedge q)$ we first compute the denotation of the subformula $\mathrm{EF}p$, by means of a least fixpoint computation, and that of q. We then compute their intersection I. Finally, we compute the denotation of the transformer AG applied to I by using a greatest fixpoint computation.

The time complexity of this model checking algorithm is polynomial in the size of the input formula φ and of the model M. It is important to notice that the number of states in the transition graph is in general exponential in the description of the model which is usually given in some high level language (e.g. a collection of formulae), and this is commonly referred to as state explosion problem. Heuristics like symbolic model checking [6] attack this problem by using compact representations of sets of states, e.g., by using binary decision diagrams as a representation of sets of states.

3 Abstract Model Checking and Refinement

Approximate automated verification by abstract model checking [9] provides one important solution to the state explosion problem [8] that arises in model checking systems with parallel components. In abstract model checking, approximation is encoded by an abstract model A that hides some details of the concrete model M so that verification becomes more efficient on A rather than on M. The design of an abstract model checking framework always includes a preservation result, roughly stating that for any formula φ expressed in some language $\mathcal{L}$, if $A \models \varphi$ then $M \models \varphi$. Clearly, abstract verification of φ on A may yield false negatives due to the approximation of M to A. On the other hand, strong preservation means that a formula φ in $\mathcal{L}$ holds on A if and only if φ holds on M. Strong preservation is thus highly desirable since it allows to draw consequences from negative answers on the abstract side.

The relationship between abstract model checking and abstract interpretation has been the subject of a number of works (e.g. [9, 19, 22, 37, 43]). We recall here how the above notion of strong preservation in abstract model checking can be generalized from an abstract interpretation perspective. This abstract interpretation-based view of strong preservation allows to understand some common principles in well-known algorithms that refine abstract Kripke structures in order to make them strongly preserving for some temporal language.

3.1 Abstract Semantics of Languages

We deal with generic (temporal) languages $\mathcal{L}$ whose state formulae φ are inductively defined by:

$$\mathcal{L} \ni \varphi ::= p \mid f(\varphi_1, ..., \varphi_n)$$

where p ranges over a (typically finite) set of atomic propositions Atoms, while f ranges over a finite set Op of operators, for example standard temporal operators like existential/universal next EX/AX, until EU/AU, globally EG/AG, etc. The semantics of a language is determined by a suitable semantic structure $\mathcal{S}$, e.g. a Kripke structure, on a concrete state space States, that provides an interpretation of atoms and operators in $\mathcal{L}$ as, respectively, elements and operators on the powerset $\wp(\mathrm{States})$. Thus, $\mathcal{S}$ determines for any formula $\varphi \in \mathcal{L}$ a concrete semantics $[\![\varphi]\!]_{\mathcal{S}} \in \wp(\mathrm{States})$, namely the set of states making φ true w.r.t. $\mathcal{S}$. In turn, this also defines a state partition $P_{\mathcal{L}} \in \mathrm{Part}(\mathrm{States})$, i.e. state equivalence, induced by the language $\mathcal{L}$ as follows:

$$P_{\mathcal{L}}(s) \triangleq \{s' \in \mathrm{States} \mid \forall \varphi \in \mathcal{L}.\, s \in [\![\varphi]\!]_{\mathcal{S}} \Leftrightarrow s' \in [\![\varphi]\!]_{\mathcal{S}}\}.$$

As shown in Section 1, abstract interpretation provides a systematic technique for approximating a concrete semantics by an abstract semantics defined on some abstract domain. We consider abstract domains of the powerset $\langle \wp(\mathrm{States}), \subseteq \rangle$ that plays here the role of concrete semantic domain. An abstract domain $A \in \mathrm{Abs}(\wp(\mathrm{States}))$, defined by abstraction/concretization maps α/γ, induces an abstract semantic structure $\mathcal{S}^A$ where the interpretation of an atom $\boldsymbol{p} \in \wp(\mathrm{States})$ is abstracted to $\alpha(\boldsymbol{p})$ while a concrete semantic operator $\boldsymbol{f} : \wp(\mathrm{States})^n \to \wp(\mathrm{States})$ is abstracted by its best correct approximation $\boldsymbol{f}^A$ on A, that is $\boldsymbol{f}^A(a_1, ..., a_n) \triangleq \alpha(\boldsymbol{f}(\gamma(a_1), ..., \gamma(a_n)))$. Thus,

any abstract domain A systematically induces an abstract semantics $[\![\varphi]\!]^A_{\mathcal{S}} \in A$ that evaluates formulae $\varphi \in \mathcal{L}$ in the abstract domain A.

It turns out that this approach based on abstract semantics generalizes standard abstract model checking [10]. Given a Kripke structure $\mathcal{K} = (\mathrm{States}, \rightarrow)$, a standard abstract model is specified as an abstract Kripke structure $\mathcal{A} = (\mathrm{AStates}, \rightarrow^\sharp)$ where the set $\mathrm{AStates}$ of abstract states is defined by a surjective map $h : \mathrm{States} \rightarrow \mathrm{AStates}$ that groups together indistinguishable concrete states whereas $\rightarrow^\sharp$ is the transition relation between abstract states. Thus, $\mathrm{AStates}$ determines a partition of States and vice versa any partition of States can be viewed as a set of abstract states.

It turns out that state partitions can be viewed as a particular class of abstract domains. On the one hand, a partition $P \in \mathrm{Part}(\mathrm{States})$ can be considered an abstract domain by means of the following Galois insertion $(\wp(\mathrm{States})_\subseteq, \alpha_P, \wp(P)_\subseteq, \gamma_P)$:

$$\alpha_P(S) \stackrel{\text{def}}{=} \{B \in P \mid B \cap S \neq \varnothing\}; \quad \gamma_P(\mathcal{B}) \stackrel{\text{def}}{=} \cup_{B \in \mathcal{B}} B.$$

Hence, $\alpha_P(S)$ encodes the minimal over-approximation of S through blocks of the state partition P. On the other hand, any abstract domain $A \in \mathrm{Abs}(\wp(\mathrm{States}))$ induces the following partition $\mathrm{part}(A) \in \mathrm{Part}(\mathrm{States})$:

$$\mathrm{part}(A)(x) \stackrel{\text{def}}{=} \{y \in \mathrm{States} \mid \alpha_A(\{y\}) = \alpha_A(\{x\})\}.$$

An abstract domain $A \in \mathrm{Abs}(\wp(\mathrm{States}))$ is called partitioning when it represents precisely a state partition, namely when $\gamma_A \circ \alpha_A = \gamma_{\mathrm{part}(A)} \circ \alpha_{\mathrm{part}(A)}$.

3.2 Generalized Strong Preservation

In standard abstract model checking, given a language $\mathcal{L}$ and a corresponding interpretation on a Kripke structure $\mathcal{K}$, an abstract Kripke structure $\mathcal{A}$ strongly preserves $\mathcal{L}$ when for any $\varphi \in \mathcal{L}$ and $s \in \mathrm{States}$, we have that

$$\mathcal{A}, h(s) \models \varphi \Leftrightarrow \mathcal{K}, s \models \varphi$$

where $h : \mathrm{States} \rightarrow \mathrm{AStates}$ is the abstraction map.

It turns out that strong preservation can be generalized from standard abstract Kripke structures to abstract interpretation-based models. A generalized abstract model is given as an abstract domain $A \in \mathrm{Abs}(\wp(\mathrm{States}))$ that systematically induces an abstract semantics $[\![\cdot]\!]^A_{\mathcal{S}}$. We therefore define the abstract semantics $[\![\cdot]\!]^A_{\mathcal{S}}$ to be strongly preserving (s.p. for short) for $\mathcal{L}$ when for any $\varphi \in \mathcal{L}$ and $S \in \wp(\mathrm{States})$,

$$\alpha(S) \leq_A [\![\varphi]\!]^A_{\mathcal{S}} \Leftrightarrow S \subseteq [\![\varphi]\!]_{\mathcal{S}}.$$

Observe that strong preservation is an abstract domain property, meaning that it does not depend on the abstract interpretation of atoms and logical/temporal operators on the abstract domain A but only depends on A itself. Thus, an abstract domain $A \in \mathrm{Abs}(\wp(\mathrm{States}))$ is strongly preserving for $\mathcal{L}$ when $[\![\cdot]\!]^A_{\mathcal{S}}$ is strongly preserving for $\mathcal{L}$.

Standard strong preservation becomes a particular instance, because it turns out that an abstract Kripke structure strongly preserves $\mathcal{L}$ if and only if the corresponding partitioning abstract domain strongly preserves $\mathcal{L}$ according to the above generalized meaning. Generalized strong preservation may work where standard strong preservation may

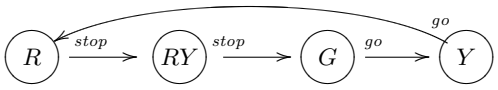

Fig. 2. A U.K. traffic light.

fail. In fact, it may happen that although a strongly preserving abstract semantics on a partition P always exists this abstract semantics cannot be derived from a strongly preserving abstract Kripke structure on P. The following example shows this phenomenon.

Example 2. Consider the following simple language $\mathcal{L}$:

$$\mathcal{L} \ni \varphi ::= \mathit{stop} \mid \mathit{go} \mid \mathrm{AXX}\varphi$$

and the Kripke structure $\mathcal{K}$ depicted in Figure 2, where superscripts determine the labeling function. $\mathcal{K}$ models a four-state traffic light controller (like in the U.K.): Red $\rightarrow$ RedYellow $\rightarrow$ Green $\rightarrow$ Yellow. According to the standard semantics of AXX, we have that $\mathcal{K}, s \models \mathrm{AXX}\varphi$ iff for any path $s_0 s_1 s_2 \ldots$ starting from $s_0 = s$, it happens that $\mathcal{K}, s_2 \models \varphi$. It turns out that $[\![\mathrm{AXX}\mathit{stop}]\!]_{\mathcal{K}} = \{G, Y\}$ and $[\![\mathrm{AXX}\mathit{go}]\!]_{\mathcal{K}} = \{R, RY\}$. We thus consider the state partition $P = \{\{R, RY\}, \{G, Y\}\}$. However, it turns out that there exists no abstract transition relation $\rightarrow^\sharp$ on the abstract state space P such that the abstract Kripke structure $\mathcal{A} = (P, \rightarrow^\sharp)$ strongly preserves $\mathcal{L}$. Assume by contradiction that such an abstract Kripke structure $\mathcal{A}$ exists. Let $B_1 = \{R, RY\} \in P$ and $B_2 = \{G, Y\} \in P$. Since $\mathcal{K}, R \models \mathrm{AXX}\mathit{go}$ and $\mathcal{K}, G \models \mathrm{AXX}\mathit{stop}$, by strong preservation, it must be that $\mathcal{A}, B_1 \models \mathrm{AXX}\mathit{go}$ and $\mathcal{A}, B_2 \models \mathrm{AXX}\mathit{stop}$. Hence, necessarily, $B_1 \rightarrow^\sharp B_2$ (otherwise B_1 can never reach the state B_2 where the atom go holds) and $B_2 \rightarrow^\sharp B_1$ (otherwise B_2 can never reach the state B_1 where the atom stop holds). This leads to the contradiction $\mathcal{A}, B_1 \not\models \mathrm{AXX}\mathit{go}$. In fact, if $\rightarrow^\sharp = \{(B_1, B_2), (B_2, B_1)\}$ then we would have that $\mathcal{A}, B_1 \not\models \mathrm{AXX}\mathit{go}$. On the other hand, if, instead, $B_1 \rightarrow^\sharp B_1$ (the case $B_2 \rightarrow^\sharp B_2$ is analogous), then we would still have that $\mathcal{A}, B_1 \not\models \mathrm{AXX}\mathit{go}$. Even more, along the same lines it is not hard to check that no proper abstract Kripke structure that strongly preserves $\mathcal{L}$ can be defined, because even if either B_1 or B_2 is split (i.e., refined) we still cannot define an abstract transition relation that is strongly preserving for $\mathcal{L}$.
On the other hand, let us consider the partitioning abstract domain

$$A \triangleq \{\varnothing, \{R, RY\}, \{G, Y\}, \{R, RY, G, Y\}\}$$

that is induced by the above partition P. This abstract domain A induces a corresponding abstract semantics $[\![\cdot]\!]^A_{\mathcal{K}} : \mathcal{L} \rightarrow A$, where the best correct approximation of the operator $\mathbf{AXX} : \wp(\mathrm{States}) \rightarrow \wp(\mathrm{States})$ on A is as follows:

$$\alpha_A \circ \mathbf{AXX} \circ \gamma_A = \{\varnothing \mapsto \varnothing, \{R, RY\} \mapsto \{G, Y\}, \{G, Y\} \mapsto \{R, RY\}, \\ \{R, RY, G, Y\} \mapsto \{R, RY, G, Y\}\}.$$

It is easy to check that this abstract semantics $[\![\cdot]\!]^A_{\mathcal{K}}$ is strongly preserving. As observed above, in the abstract Kripke structure $\mathcal{A}$, the formulae $\mathrm{AXX}\mathit{go}$ and $\mathrm{AXX}\mathit{stop}$ are not strongly preserved. Here, instead, we have that $\alpha_P(S) \leq_A [\![\mathrm{AXX}\mathit{go}]\!]^A_{\mathcal{K}} \Leftrightarrow S \subseteq [\![\mathrm{AXX}\mathit{go}]\!]_{\mathcal{K}}$ and $\alpha_P(S) \leq_A [\![\mathrm{AXX}\mathit{stop}]\!]^A_{\mathcal{K}} \Leftrightarrow S \subseteq [\![\mathrm{AXX}\mathit{stop}]\!]_{\mathcal{K}}$. □

3.3 Strong Preservation as Completeness

Given a language $\mathcal{L}$ and a Kripke structure $\mathcal{K} = (\mathrm{States}, \rightarrow)$, a well-known key problem is to compute the smallest abstract state space $\mathrm{AStates}_{\mathcal{L}}$, when this exists, such that one can define an abstract Kripke structure $\mathcal{A}_{\mathcal{L}} = (\mathrm{AStates}_{\mathcal{L}}, \rightarrow^{\sharp})$ that strongly preserves $\mathcal{L}$. This problem admits solution for a number of well-known temporal languages like CTL (or, equivalently, the μ-calculus), ACTL and CTL-X (i.e. CTL without the next-time operator X). A number of algorithms for solving this problem exist, like those by Paige and Tarjan [61] for CTL, by Henzinger et al. [49], Tan and Cleaveland [74], Ranzato and Tapparo [65] and Gentilini et al. [32, 47] for ACTL, and Groote and Vaandrager [48] for CTL-X. These are coarsest partition refinement algorithms. Given a language $\mathcal{L}$ and a state partition $P \in \mathrm{Part}(\mathrm{States})$ which is determined by a state labeling $\ell : \mathrm{States} \rightarrow \wp(\mathrm{Atoms})$ — namely, $P \triangleq \{\ell^{-1}(X) \mid X \subseteq \mathrm{Atoms}\}$ — these algorithms can be viewed as computing the coarsest partition $P_{\mathcal{L}}$ that refines P and allows to define an abstract Kripke structure $(P, \rightarrow^{\sharp})$ that strongly preserves $\mathcal{L}$. It is worth remarking that most of these algorithms have been designed for computing well-known behavioural equivalences used in process algebra like bisimulation (for CTL), simulation (for ACTL) and divergence-blind stuttering (for CTL-X) equivalence. Our abstract interpretation-based framework allows us to provide a generalized view of these partition refinement algorithms. It turns out that the most abstract (i.e., least informative) domain, denoted by $\mathrm{AD}_{\mathcal{L}}$, that strongly preserves a given language $\mathcal{L}$ always exists, namely the domain

$$\sqcup\{A \in \mathrm{Abs}(\wp(\Sigma)) \mid A \text{ is s.p. for } \mathcal{L}\}$$

results to be s.p. for $\mathcal{L}$. It turns out that $\mathrm{AD}_{\mathcal{L}}$ is a partitioning abstract domain if and only if $\mathcal{L}$ includes propositional logic, that is when $\mathcal{L}$ is closed under logical conjunction and negation. Otherwise, a proper loss of information occurs when abstracting $\mathrm{AD}_{\mathcal{L}}$ to the corresponding partition $P_{\mathcal{L}}$. Moreover, for some languages $\mathcal{L}$, it may happen that one cannot define an abstract Kripke structure on the abstract state space $P_{\mathcal{L}}$ that strongly preserves $\mathcal{L}$ whereas the most abstract strongly preserving domain instead exists. In fact, in Example 2, the domain A actually is the most abstract s.p. domain for the language $\mathcal{L}$ whilst no s.p. abstract Kripke structure can be defined.

As discussed in Section 1, completeness in abstract interpretation encodes an ideal situation where the abstract semantics coincides with the abstraction of the concrete semantics. A precise correspondence between generalized strong preservation and completeness in abstract interpretation can be established. This is based on the notion of forward complete abstract domain. As recalled in Section 1, it turns out that forward complete abstract domains can be systematically and constructively derived from noncomplete abstract domains by minimal refinements. Given any domain $A \in \mathrm{Abs}(C)$, recall that we denote by $\mathcal{F}\text{-}\mathrm{Shell}_{\boldsymbol{f}}(A)$ the forward complete shell of A for $\boldsymbol{f}$. $\mathcal{F}\text{-}\mathrm{Shell}_{\boldsymbol{f}}(A)$ can be obtained by iteratively closing $\gamma(A)$ under direct images of $\boldsymbol{f}$ until a fixpoint is reached, i.e.,

$$\mathcal{F}\text{-}\mathrm{Shell}_{\boldsymbol{f}}(A) \triangleq \mathrm{lfp}\left(\lambda X \subseteq C.\gamma(A) \cup X \cup \boldsymbol{f}(X)\right).$$

It turns out that strong preservation is related to forward completeness as follows. As described above, the most abstract domain $\mathrm{AD}_{\mathcal{L}}$ that strongly preserves $\mathcal{L}$ always exists.

It turns out that $\mathrm{AD}_{\mathcal{L}}$ coincides with the forward complete shell for the logical/temporal operators of $\mathcal{L}$ of a basic abstract domain $A_\ell \triangleq \mathcal{M}(\{\ell^{-1}(X) \mid X \subseteq \mathrm{Atoms}\})$ determined by the state labeling ℓ, i.e.,

$$\mathrm{AD}_{\mathcal{L}} = \mathcal{F}\text{-}\,\mathrm{Shell}_{\mathbf{Op}_{\mathcal{L}}}(A_\ell).$$

This characterization provides a generalization of partition refinement algorithms used in standard abstract model checking that can be therefore logically viewed as refinements w.r.t. forward completeness.

Example 3. Conside the above Example 2 where the labeling determines the abstract domain $A_\ell = \{\varnothing, \{R, RY\}, \{G, Y\}, \{R, RY, G, Y\}\}$. Let $\mathbf{AXX}$ be the semantic interpretation of AXX. It turns out that A_ℓ is already forward complete for $\mathbf{AXX}$ because $\mathbf{AXX}(\{R, RY\}) = \{G, Y\}$ and $\mathbf{AXX}(\{G, Y\}) = \{R, RY\}$. Thus, here

$$\mathrm{AD}_{\mathcal{L}} = \mathcal{F}\text{-}\,\mathrm{Shell}_{\mathbf{AXX}}(A_\ell) = A_\ell$$

namely A_ℓ is the most abstract strongly preserving domain for the language $\mathcal{L}$. □

Bisimulation Equivalence. As an example, let us describe how this approach allows us to derive a novel characterization of bisimulation equivalence in terms of forward completeness of abstract domains.

Bisimulation equivalence P_{bis} on some Kripke structure $\mathcal{K}$ can be computed by the well-known Paige-Tarjan partition refinement algorithm PT. More precisely, if P_ℓ denotes the state partition determined by the labeling function ℓ then $\mathrm{PT}(P_\ell) = P_{\mathrm{bis}}$. It is well known [5] that when $\mathcal{K}$ is finitely branching, bisimulation equivalence coincides with the state equivalence induced by Hennessy-Milner logic

$$\mathrm{HML} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EX}\varphi$$

that is, $P_{\mathrm{HML}} = P_{\mathrm{bis}}$. As usual, the semantic interpretation of EX is the predecessor $Pre : \wp(\mathrm{States}) \to \wp(\mathrm{States})$, while conjunction and negation are, respectively, interpreted as intersection $\cap$ and complementation $\complement$ on $\wp(\mathrm{States})$.

The following characterization can then be derived in our abstract interpretation-based framework:

$$\mathrm{PT}(P_\ell) = \mathrm{part}(\mathcal{F}\text{-}\,\mathrm{Shell}_{\{Pre, \complement\}}(A_\ell)).$$

Note that the forward complete shell does not need to take into account the intersection on $\wp(\mathrm{States})$ since abstract domains, being closed under intersections, are always forward complete for intersections. This characterization in turn leads to design a generalized Paige-Tarjan-like procedure for computing most abstract strongly preserving domains [66].

4 Model Checking and (Constraint) Logic Programming

4.1 Model Checking and Fixpoint Semantics in LP

As discussed in Section 2, the semantics of CTL properties is defined as a least or greatest fixpoint of a monotonic operator defined over sets of configurations, i.e., states.

This property can be exploited in order to provide a link between model checking and logic programming. As an example, let us interpret an atomic formula $p(s_1, s_2, val)$ as a configuration of a system with two processes whose current states are, resp., s_1 and s_2 and with a shared variable whose current value is val. Now let P be the logic program defined as

$$\begin{array}{l} p(idle, X, free) : -p(use, X, lock). \\ p(use, X, Y) : -p(idle, X, free). \\ p(X, idle, free) : -p(X, use, lock). \\ p(X, use, Y) : -p(X, idle, free). \end{array}$$

According to the above mentioned interpretation of the predicate p, the Horn clauses in P represent one-step transitions (possible moves of one of the two processes) of a concurrent system in which the access to the critical section use is controlled via modifications to the global variable with states $lock$ and $free$.

Let us now consider the set of ground atomic predicates

$$Bad \triangleq \{p(use, use, lock),\ p(use, use, free)\}.$$

They represent violations to the mutual exclusion property for the system represented by the program P. To draw a link between the semantics of P and CTL properties like EF, we need to resort to the fixpoint semantics of logic programs. We first recall that the immediate consequence operator of the logic program $Q \triangleq P \cup Bad$ is defined as

$$T_Q(I) \triangleq \{A\theta \mid A : -B \in P,\ B\theta \in I,\ \theta \text{ grounding for } A, B\} \cup Bad$$

where I is a set of ground atoms with predicate p and constants taken from the set $\{idle, busy, free, lock\}$. It is immediate to see that when T_Q is applied to a set of atoms I, it computes (a representation of) the set of one-step predecessors of the configurations in I. The fixpoint semantics $\mathcal{F}_Q$ of the program Q is defined as the least fixpoint of the T_Q operator, i.e., as the set of ground atoms

$$\mathcal{F}_Q \triangleq \text{lfp}(T_Q) = \bigcup_{i \geq 0} T_Q^i(\varnothing).$$

Based on the link between T_Q and the operator Pre used in the semantics of CTL, we have that $\mathcal{F}_Q$ is a representation of the set of all predecessors of violations to mutual exclusion contained in Bad. In other words, $\mathcal{F}_Q$ is equivalent to the denotation of the CTL formula $\text{EF}(use_1 \wedge use_2)$, where use_i is the predicate that is true if and only if the process i is in the critical section. In a similar way, we can use the greatest fixpoint semantics of logic programs to characterize CTL properties like EG.

4.2 From Finite-state to Infinite-state Models

The interpretation of logic programs as a symbolic representation of transition systems paves the way to several different logic-based methods for the verification of finite-state and infinite-state systems. For instance, in [24], the s-semantics of constraint logic programs is applied to symbolically reason on infinite-state transition systems. The s-semantics of logic programs is obtained by lifting the fixpoint semantics to a domain in

which interpretations are sets of nonground atoms. Going back to the previous example, we first observe that the set Bad can be represented with the single nonground atom

$$b \triangleq p(use, use, X)$$

where X is a free variable. Furthermore, the bottom-up evaluation of the program $Q' \triangleq P \cup \{b\}$ can be computed symbolically by replacing the operator T_Q with the corresponding nonground version S_Q. The nonground immediate consequence operator S_Q is obtained by replacing in the definition of T_Q the grounding substitution θ with the most general unifier between B and an atom in I. More formally, given a set of nonground atoms I, the operator S_Q is defined as

$$S_Q(I) \triangleq \{A\theta \mid A : -B \in P,\ C \in I,\ \theta = \text{m.g.u.}(B, C)\} \cup Bad.$$

The nonground fixpoint semantics is defined as the least fixpoint of the S_Q operator, i.e., as the result of a bottom-up evaluation of Q. It is important to notice that the subsumption test between nonground atoms can be used as termination test for this type of symbolic fixpoint computation. Optimizations like magic set templates can be used to specialize the bottom-up evaluation procedure with respect to a given query (e.g., a set of initial states).

As shown in [24], the s-semantics for CLP can be used to extend the link between bottom-up evaluation of logic programs and model checking to the case of infinite-state transition systems. CLP clauses can be used to symbolically represent a possibly infinite set of transition rules, and constrained atoms, i.e., atoms like $p(X, Y) : -X > Y$ can be used to symbolically represent infinite sets of configurations, i.e., all the instances obtained by solving the constraint.

4.3 Verification and Evaluation Strategies in LP

Several other types of evaluation of logic programs have been proposed for the verification of temporal properties of transition systems.

In [30, 31] the transition system of counter automata (automata with guards and assignments over a finite set of counters) are symbolically represented as logic programs with linear arithmetic constraints. The bottom-up evaluation of logic programs with gap-order constraints (obtained by relaxing the linear constraints in the automata) is used to over-approximate the set of successors, i.e., $Post^*$, of the original automata.

Program specialization methods (e.g. partial evaluation) is another example of techniques that can be used to automatically control the abstraction required for infinite-state model checking [54, 52, 53]. In [29, 62] program transformation techniques combined with specialized decision procedures are used to verify temporal properties of infinite-state systems.

The application of tabling to the evaluation of logic programs represents a further important research line in-between logic programming and verification. The model checker XMC based on the XSB system has been applied to several families of verification problems and concurrent models including pi-calculus and mobile process algebra [25, 64, 68, 69, 71]. For this kind of systems, tabling can be used to efficiently evaluate

logic programs that encode the semantics of CTL operators. Since tabling exploits different types of subsumption mechanisms, the resulting engine can be applied both to finite-state and infinite-state systems.

As shown in [4, 23], bottom-up evaluation methods for logic programming languages like LO [1] and MSR [7] extend the use of symbolic techniques based on unification (e.g. S_P-like operators) to languages that naturally model concurrency via multiset rewriting.

Other promising approaches to verification are based on logic programming languages that incorporate connectives to express least and greatest fixpoint computations as Bedwyr [2] and LolliMon [58]. These languages are based on nonstandard logics like intuitionistic and linear logic and can be used to specify operational semantics and temporal properties of a large class of concurrent systems.

References

1. J.-M. Andreoli, R. Pareschi: Linear Ojects. Logical Processes with Built-in Inheritance. *New Generation Comput.*, 9(3/4): 445-474 (1991)
2. D. Baelde, A. Gacek, D. Miller, G. Nadathur, A. Tiu. The Bedwyr system for model checking over syntactic expressions In *Proc. CADE 2007*: 391-397.
3. R. Bagnara, P. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theor. Comput. Sci.*, 277(1-2):3–46, 2002.
4. M. Bozzano, G. Delzanno, M. Martelli. Model Checking Linear Logic Specifications. *TPLP*, 4(5-6): 573-619 (2004)
5. M.C. Browne, E.M. Clarke and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoret. Comp. Sci.*, 59:115–131, 1988.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: 10^20 States and Beyond In *Proc. IEEE LICS 1990*:428-439, 1990.
7. I. Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. *ENTCS* 40, 2000.
8. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back, 10 Years Ahead*. LNCS 2000, pp. 176-194, 2001.
9. E.M. Clarke, O. Grumberg and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
10. E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. The MIT Press, 1999.
11. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.
12. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Sci. Comput. Program.*, 38(1-3):27–71, 2000.
13. P. Cousot. Types as abstract interpretations (Invited Paper). In *Proc. ACM POPL'97*, pp. 316–331, 1997.
14. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pp. 238–252, 1977. ACM Press.

16. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pp. 269–282, 1979. ACM Press.
17. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Program.*, 13(2-3):103–179, 1992.
18. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) (Invited Paper). In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '94*), pp. 95–112, 1994.
19. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. 27th ACM POPL*, pp. 12–25, 2000.
20. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proc. ESOP'05*, LNCS 3444, pp. 21–30, 2005.
21. P. Cousot, R. Cousot, and R. Giacobazzi. Abstract interpretation of resolution-based semantics. *Theor. Comput. Sci.*, 410(46):4724–4746, 2009.
22. D. Dams, O. Grumberg and R. Gerth. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1997.
23. G. Delzanno. An Overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems. *ENTCS* vol. 76, 2002.
24. G. Delzanno, A. Podelski. Model Checking in CLP. In *Proc. TACAS 1999*: 223-239, 1999.
25. Y. Dong, X. Du, Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, O. Sokolsky, E.W. Stark, D. Scott Warren. Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools. In *Proc. TACAS 1999*, pp. 74-88, 1999.
26. M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *Theor. Comput. Sci.*, 69(3):289–318, 1989.
27. G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.
28. G. Filé and F. Ranzato. Complementation of abstract domains made easy. In *Proc. of the 1996 Joint Internat. Conf. and Symp. on Logic Programming* (*JICSLP '96*), pp. 348–362, 1996.
29. F. Fioravanti, A. Pettorossi, M. Proietti. Verification of Sets of Infinite State Processes Using Program Transformation. In *Proc. LOPSTR 2001*, pp. 111-128, 2001.
30. L. Fribourg, J. Richardson. Symbolic Verification with Gap-Order Constraints. In *Proc. LOPSTR 1996*, pp. 20-37, 1996.
31. L. Fribourg, H. Olsén. A Decompositional Approach for Computing Least Fixed-Points of Datalog Programs with Z-Counters. *Constraints*, 2(3/4):305-335, 1997.
32. R. Gentilini, C. Piazza and A. Policriti. From bisimulation to simulation: coarsest partition problems. *J. Automated Reasoning*, 31(1):73-103, 2003.
33. R. Giacobazzi and I. Mastroeni. Compositionality in the puzzle of semantics. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pp. 87–97, 2002.
34. R. Giacobazzi and I. Mastroeni. Transforming semantics by abstract interpretation. *Theor. Comput. Sci.*, 337(1-3):1–50, 2005.
35. R. Giacobazzi and I. Mastroeni. Transforming abstract interpretations by abstract interpretation. In *Proc. 15th International Static Analysis Symposium, SAS'08*, LNCS 5079, pp. 1–17, 2008.
36. R. Giacobazzi, C. Palamidessi, and F. Ranzato. Weak relative pseudo-complements of closure operators. *Algebra Universalis*, 36(3):405–412, 1996.
37. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In *Proc. 8th Internat. Static Analysis Symp. (SAS'01)*, LNCS 2126, pp. 356–373, 2001.

38. R. Giacobazzi and F. Ranzato. Complementing logic program semantics. In *Proc. of the 5th Internat. Conf. on Algebraic and Logic Programming* (*ALP '96*), LNCS, pp. 238–253, 1996.
39. R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Proc. ICALP'97*, LNCS 1256, pp. 771–781, 1997.
40. R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program*, 32(1-3):177–210, 1998.
41. R. Giacobazzi and F. Ranzato. Uniform closures: order-theoretically reconstructing logic program semantics and abstract domain refinements. *Information and Computation*, 145(2):153–190, 1998.
42. R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theor. Comput. Sci*, 216:159–211, 1999.
43. R. Giacobazzi and F. Ranzato. Incompleteness of states w.r.t. traces in model checking. *Information and Computation*, 204(3):376-407, 2006.
44. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
45. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract domains condensing. *ACM Transactions on Computational Logic*, 6(1):33–60, 2005.
46. R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5):1067–1109, 1998.
47. R. van Glabbeek and B. Ploeger. Correcting a space-efficient simulation algorithm. In *Proc. 20^{th} CAV*, LNCS 5123, pp. 517-529, 2008.
48. J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17^{th} ICALP*, LNCS 443, pp. 626-638, 1990.
49. M.R. Henzinger, T.A. Henzinger and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36^{th} FOCS*, pp. 453-462, 1995.
50. T.A. Henzinger, R. Maujumdar and J.-F. Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Log.*, 6(1):1-31, 2005.
51. T. P. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. Program. Lang. Syst.*, 19(5):751–803, 1997.
52. M. Leuschel and H. Lehmann. Coverability of Reset Petri Nets and Other Well-Structured Transition Systems by Partial Deduction. In *Proc. Computational Logic 2000*, pp. 101-115
53. M. Leuschel and H. Lehmann. Solving coverability problems of petri nets by partial deduction. In *Proc. PPDP 2000*, pp. 268-279, 2000.
54. M. Leuschel and T. Massart. Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In *Proc. LOPSTR 1999*, pp. 62-81, 1999.
55. G. Levi and F. Spoto. An experiment in domain refinement: Type domains and type representations for logic programs. In *Proc. PLILP/ALP'98*, LNCS 1490, pp. 152–169, 1998.
56. G. Levi and F. Spoto. Non pair-sharing and freeness analysis through linear refinement. In *Proc. ACM PEPM*, pp. 52–61, 2000.
57. G. Levi and F. Spoto. Pair-independence and freeness analysis through linear refinement. *Information and Computation*, 182(1):14–52, 2003.
58. P. López, F. Pfenning, J. Polakow, K. Watkins. Monadic concurrent linear logic programming. In *Proc. PPDP 2005*, pp. 35-46, 2005.
59. A. Mycroft. Completeness and predicate-based abstract interpretation. In *Proc. of the ACM Symp. on Partial Evaluation and Program Manipulation* (*PEPM '93*), pp. 179–185, 1993.
60. F. Nielson. Expected forms of data flow analyses. In *Proceedings of Programs as Data Objects Workshop*, LNCS 217, pp. 172–191, 1986.
61. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):977–982, 1987.
62. A. Pettorossi, M. Proietti, V. Senni. Transformational Verification of Parameterized Protocols Using Array Formulas. In *Proc. LOPSTR 2005*, pp. 23-43, 2005.

63. C.R. Ramakrishnan. A Model Checker for Value-Passing Mu-Calculus Using Logic Programming. In *Proc. PADL 2001*, pp. 1-13, 2001.
64. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Proc. CAV 1997*, pp. 143-154, 1997.
65. F. Ranzato and F. Tapparo. A new efficient simulation equivalence algorithm. In *Proc. 22nd IEEE Symp. on Logic in Computer Science (LICS'07)*, pp. 171–180, 2007.
66. F. Ranzato and F. Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Information and Computation*, 206(5):620-651, 2008.
67. K.I. Rosenthal. Quantales and their applications. In *Pitman Research Notes in Mathematics*. Longman Scientific & Technical, London, 1990.
68. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka. Verification of Parameterized Systems Using Logic Program Transformations. In *Proc. TACAS 2000*, pp. 172-187, 2000.
69. A. Roychoudhury, C.R. Ramakrishnan. Unfold/Fold Transformations for Automated Verification of Parameterized Concurrent Systems. In *Program Development in Computational Logic 2004*: 261-290, 2004.
70. F. Scozzari. Logical optimality of groundness analysis. *Theor. Comput. Sci.*, 277(1-2):149–184, 2002.
71. A. Singh, C.R. Ramakrishnan, S.A. Smolka. Query-Based Model Checking of Ad Hoc Network Protocols. In *Proc. CONCUR 2009*, pp. 603-619, 2009.
72. F. Spoto. Optimality and condensing of information flow through linear refinement. *Theor. Comput. Sci.*, 388(1-3):53–82, 2007.
73. C. Strachey. The varieties of programming language. In *Proc. of the International Computing Symposium*, pp. 222–233, Cini Foundation, Venice, Springer, 1972.
74. L. Tan and R. Cleaveland. Simulation revisited. In In *Proc. TACAS'01*, LNCS 2031, pp. 480-495, 2001.
75. P. Yang, S. Basu, C. R. Ramakrishnan. Parameterized Verification of pi-Calculus Systems. In *Proc. TACAS 2006*, pp. 42-57, 2006.