1.1.1 Efficient Verification based on abstraction (1991-1998)

So far, we had mainly concentrated on optimisation methods computing state graph reductions modulo some equivalence, and which *strongly preserves* some set of properties of interest and allows exploiting both *correctness proofs* and *error traces*. Nevertheless, relaxing this criterion makes the computation of reduced models easier and allows using smaller models.

Abstract interpretation [CC77, CC80] is a general framework for abstractions which preserve extreme fixpoints of monotonic functions between property lattices. In this framework *properties* are sets of configurations or states, that is state predicates. Property lattices of interest may be the boolean algebra induced by a set of states or predicates, but they may be arbitrary complete lattices.

1.1.1.1 Property preservation by abstraction

In [LGS+95], we study property preserving simulation-based abstractions. We define a notion of (α, γ)-simulation, which is a simulation parameterized by a Galois connexion (α, γ) between a concrete and an abstract property lattice. We establish the relationship between abstract interpretation and simulation by showing that this notion of Galois connection induced simulation coincides with Milner's notion of ρ-simulation [Mil71] by defining for each Galois connection a corresponding simulation relation and vice versa.

We also establish results of weak and strong property preservation for universal and existential fragments of the μ-calculus between transition systems related by (α, γ)-simulations.

We show also that under some minimal sufficient conditions, abstractions of a system can be computed by composing abstractions of its components, and this for any of the usual parallel composition operators: when the abstractions of components do not abstract the interfaces between the components, then abstraction and parallel composition commute, otherwise abstracting before composition yields a weaker abstraction (see also [GL93a]). This result represents a generalisation of the property known for CCS which allows the application of label abstraction and weak bisimulation reduction before parallel composition, as long as no labels of the interface are renamed (see also section **??**)).

This framework for abstraction extends those proposed in [CGL92] — where abstractions are restricted to homomorphisms from the concrete to the abstract state space — and in [Kur87] — which also considers homomorphism-based abstractions and which formulates preservation results for language inclusion only.

It has later been extended by using a pair of abstraction relations (or Galois connections), one overapproximating the set of successors and one undervapproximating it [KDG95], so as to obtain preservation results for the full μ-calculus.

Based on this framework, we have defined several concrete frameworks for effectively computing abstractions

1.1.1.2 Computation of abstractions

1.1.1.2.1 Calculating abstractions of Boolean programs In the context of the thesis work of Claire Loiseaux [Loi94], we have used this theory to compositionally compute abstractions of *Boolean transition systems* defined by a set of transitions represented by *guarded commands* on Boolean variables.

(α, γ)-simulations are defined by an expression defining a relation ρ between concrete and abstract states, that is by posing $\alpha = post[\rho]$ and $\gamma = \widetilde{pre}[\rho]$ which are quantified expressions that can be computed effectively for boolean expressions.

Notice that the exact abstraction of an individual command $\alpha \circ pre[com] \circ \gamma$ is not always of the form $pre[com^A]$ — that is a precondition of a transition relation. There may be several minimal transition relations com^A. In [LGS+95], we have defined a notion of faithfulness, defining a criterion of minimality up to bisimulation, and we have shown that whenever ρ is total and $\rho \circ \rho^{-1}$ is a closure, then the transition relation defined by $com^A = \rho \circ com \circ \rho^{-1}$ is faithful. To compute an abstract program, we replace each guarded command (transition relation) com by an abstraction com^A.

We have implemented a tool, called *Oscar* (see section **??**) that uses BDDs[1] for the encoding and manipulation of boolean expressions, which constructs abstractions of boolean programs and evaluates formulas of a univesal fragment of temporal logic.

Using this prototype, we have verified a set of safety properties of non trivial examples, in particular a token ring protocol and a non trivial toy service in the context of an automated highway, managing overtaking manoeuvres [GL93c, GL93a, GL93b]. In the second example, the compositional approach to abstraction was essential.

1.1.1.2.2 Use of abstract Data types for computing abstractions and verification of a Cache memory system Abstract interpretation [CC77] and also the approach chosen in [CGL94] propose an even more compositional approach. They do not aim at building exact abstractions of entire guarded commands, but compose abstractions for atomic functions and predicates. The obtained abstract programs are generally less precise than the ones that we compute, nevertheless it is often much easier to compute: for finite domains it implies computation on smaller sets of variables and for infinite domains it allows building libraries of useful abstractions of particular data types and their associated predicates and functions.

In [Gra94, Gra99], such an approach is used for the validation of Lotos specifications, where data types and their functions are represented by abstract data types. In this case, an "abstract interpretation" is obtained simply by replacing the original (concrete) abstract data type of a given – structured – variable by a more abstract one.

This method has been applied to a distributed memory system with lazy caching, where *read* and *write* requests are propagated via a system of lossy buffers [ABM93]. The aim is to verify that this system has the property of *Sequential consistency* [Lam79], that is, it guarantees that in any single location, the order of *read* and *write* actions executed is compatible with a global sequential memory. This system is infinite, as the number of participating nodes and the size of request queues may be unlimited. In the context of the REACT project, several verification methods have been proposed [Ger99]. All approaches suppose without loss of generality that each datum D is written at most once at the same address A.

The paper [Gra99] contains several results which could be interesting beyond the mere case study, for the verification of other cache memory systems, or more generally systems communicating through buffers.

- Different kinds of buffer abstractions where needed which distinguish by their *put* and *get* methods. They are more precise than the usual counter abstractions, such as those proposed in [Cri95] as they are property oriented.

- We succeeded in verifying the cache memory for an arbitrary number of cache lines (called *nodes* hereafter). Indeed, (depending on the property to be verified) the abstract system consisting of n nodes of type T, is defined as a composition of the same number of abstract nodes, of type T_i^A. In each abstraction only a small number of the types T_i^A are distinct, in particular there exists at least one node type T_{gen}^A representing a memoryless abstraction, such that, independently of n, all but a fixed number of nodes are abstracted to one of these T_{gen}^A-types. This guarantees for any parallel composition that T_{gen}^A is an invariant of the set of nodes representing the "noise" of the rest of the system with respect to the property to be verified and the composition of a small number of abstract nodes defines a network invariant (see [WL89]).

  Some time later, we have proposed in [GLW99] a generalisation of this abstraction method, where the types T_{gen}^A may be memoryful.

- A third result of this study is a characterisation of the notion of sequential consistency which is "exact enough in practise".

It was known that the notion of sequential consistency cannot be characterized exactly in CTL by a formula depending only on the observable *events read* and *write* as in Lamport's definition saying that

[1] Binary Decision Diagrams [Bry86]

the result of any execution is the same as if the operations [memory accesses read and write] of all processors were executed in some sequential order, and the operations of each individual processor appear in in this sequence in the order specified by its program.

Lamport's characterisation can be used directly to define a tester which checks the safety property implied by the above property, that is that for each observed prefix if an appropriate ordering is possible. We are interested in a temporal logic characterization allowing to to *verify* any given algorithm and not any individual execution.

In the paper, we have given a set of temporal properties which implies sequential consistency. I was convinced that this characterisation comes indeed close enough to an exact characterisation to be applicable to any reasonable algorithm. However, taking up this characterisation several years later, I believe that it could be better structured and simplified. It could also be weakened a bit in order to allow algorithms with justified "anticipating reads" which I had considered to be a priori unrealistic.

1.1.1.3 Use of constraint solvers for model-checking and predicate abstractions

Motivated by the fact that our tools did not allow constructing the abstractions used for the verification of the cache memory system, even if the abstraction relation has been fixed, we were interested in extended finite state abstraction and verification methods to systems with unbounded data types. In the following we suppose that safety properties are expressed as invariants that the system Sys must have. The semantics of a system is represented by a transition τ which will be defined in a more structured form. For φ a predicate representing a set of states, $\Box\varphi$ represents the greatest fixpoint of $F(X) = X \cap \widetilde{pre}[\tau](X)$, a function on the property lattice under consideration. Therefore, we represent invariants by properties of the form $\Box\varphi$ and the fact that the system Sys has this invariant by

$$Sys \models init \Rightarrow \Box\varphi \quad (1)$$

The standard method for proving (1) requires coming up with an *inductif* invariant stronger than φ, that is a predicate P satisfying

$$P \Rightarrow \varphi \quad \wedge \quad P \Rightarrow \widetilde{pre}[\tau](P) \quad (2)$$

A possible model-checking technique for (1) consists in *computing* the greatest P satisfying (2) as the limit of the decreasing chain of properties

$$P^0 = \varphi\,;\ P^{i+1} = P^i \cap \widetilde{pre}[\tau](P^i) (3)$$

In order to be able to apply this "truly symbolic" model-checking technique, we need a representation of the elements of the property lattice for which it is possible to represent or compute the function F — that is boolean operations and the function $\widetilde{pre}[\tau]$ — and for which there is a (semi-) decision procedure for proving implications of the form (2), that is termination of the algorithm. The existence of a fixpoint can be guaranteed if the length of the decreasing chains calculated in (3) is always finite, that is, if the relevant subset of the property lattice is well-founded. If only a semi-decision procedure exists, then it is possible that termination cannot be detected, even if the fixpoint is reached.

In the context of the thesis of Hassen Saïdi [Sai98], we decided to use the theorem prover PVS [SOR93] to discharge the verification conditions requested by this proof rule. PVS includes a decision procedure for a useful subset of Presburger arithmetic. The interactive theorem prover of PVS can also be used as a semi-decision procedure for richer theories using a language for the definition of complex tactics which can be used as functions with a result of type "proof" or "no proof found".

For the structured representation of *Symbolic transition systems*, we extend the guarded command language on boolean variables of section 1.1.1.2 to the rich type system of PVS which includes subtyping and inductive types.

We built a tool, *Invariant-checker* using PVS to implement the model-checking algorithm above on parallel compositions of *Symbolic transition systems* [GS96]. Notice that PVS is only used as a "solver", to discharge verification conditions, and not to encode the program semantics. We use PVS however for generating type checking conditions associated with guarded commands which – if they cannot be automatically discharged – are invariants to be proven.

The tool implements a proof tactic combining boolean simplifications, rewriting strategies, decision procedures, inductive proofs for theorems on inductive types as well as a user definable set of auxiliary theorems which are often necessary.

But used in a naive way — even exploiting the fact that $\widetilde{pre}[\tau](P)$ can be calculated compositionally for τ a union of transition relations and that (2) can be checked individually for each disjunct of τ — this model-checking procedure turns out to be unusable in practise if the fixpoint computation takes more than a few iterations to converge. This is due to the fast growth of the size of the expressions representing approximants.

Different tactics have been envisaged to try to either accelerate convergence or avoid the fast growth of terms. Our first idea was to apply simplification procedures for first order formulas. It turned out however that there existed no such tool doing significantly better than the rewriting heuristics already implemented in PVS.

The STeP temporal logic prover [MBB+95] and TPVS [BLUP94] apply a similar strategy as ours applying appropriate proof rules for arbitrary temporal formula. The more recent tool TLPVS [PA03] handles also liveness properties. Finding a proof corresponds to refining (unfolding) the minimal model of the formula until it matches the symbolic transition system to be analysed. STeP asks the user to interactively choose some tactic for the next refinement step, and then automatically applies it to the current model.

In [HS96] a means for avoiding the growth of the size of formula is proposed: if Q must be shown an invariant, but is not inductive, then $Q \Rightarrow \widetilde{pre}[\tau](Q)$ does not reduce to $true$, but to some formula R. Then, one may check in the following step the invariance of R instead of the invariance of $Q \cap \widetilde{pre}[\tau](Q)$, as R seems often to be simpler. This method does however not help accelerating the convergence.

The tool CAVEAT [GR95] proposes the use of auxiliary invariants to improve interactive proofs of invariants. This tools proposes also some strategies for efficiently exploiting large repositories of potential invariants efficiently. But CAVEAT does no automatic strengthening. In our tool, we use the strengthening with automatically generated structural invariants as proposed in [BLS96].

In some cases, this method is very successful. Nevertheless, the structural invariants generated by the methods of [BLS96] are compositional invariants, which makes them easy to construct. They eliminate only non reachable states where the state of each component can be proven unreachable independently of the state of the environment. Global control invariants are however often the key to the verification of global properties.

In order to get better global control invariants, we propose in [GS97] an abstraction method, later called predicate abstraction. For a set of predicates p_i which are considered useful as a basis for the construction of a an abstract global control graph, we construct a boolean transition system on a set of boolean variables B_i, such that $B_i = true$ represents the set of concrete states in which the predicate p_i holds. This defines a Galois connection (α, γ), where γ can be easily computed for an arbitrary expression by replacing each occurrence of a variable B_i by the corresponding concrete predicate p_i, and where α is only applied to boolean expressions exp having p_i as variables, for which the evaluation of $\alpha(exp)$ is trivial.

Instead of calculating for any set of abstract states $exp^A(B_1...B_n)$ its successor set by applying $\gamma \circ post[\tau] \circ \alpha$ which may lead to non reducable, and thus non usable expressions, we rather try to prove for some set of states $exp^{A'}$ that $exp^{A'}$ is unreachable from exp^A via τ, by proving (with the proof strategies of the invariant checker) that

$$exp^A \Rightarrow \widetilde{pre}[\tau](\neg exp^{A'})$$

holds.

Such proof steps are combined so as to compute an abstraction of the control graph defined by $B_1, ...B_n$. Notice that the most precise naive approach would consist in analysing for each $2^n \times 2^n$ abstract state pairs if there exists an abstract transition between them. In [GS97] we propose the following strategy which combines overapproximation, reachability analysis and simplifications based on static dependency analysis. We suppose that τ is of the form $\vee \tau_i$. Start by a static dependency analysis amongst the predicates p_i to detect infeasible valuations of the abstract state. Then try to establish some trivial facts about the abstract transition relations τ_i^A; in particular, try to find for τ_i a set of B_j that are independent

of τ_i and thus never change by the application of τ_i. Or find a set of B_j set to a fixed value (0 or 1) by the application of τ_i. These facts defines a first overapproximation τ_i^{A0} of τ_i that is simple to compute.

The next step is to further strengthen the relations τ_i^{A0} by a combined reachability analysis and successor elimination. For those abstract states S^A which cannot be proven unreachable, try to find a set of successor states via τ_i^{A0} which can be proven unreachable via τ_i, and strengthen the relations τ_i^{A0} accordingly. As each abstract state has potentially 2^n successors (if we have already proven some facts about τ_i this set may be considerably smaller), we may not be able to examine them individually. As a heuristics – reducing the number of proofs for each set of start states from 2^n to $2 * n$, we choose to consider only successor sets defined by monomials on B_i. The reason is, that this can be done by at most $2 \times n$ proofs trying to establish for each abstract variable B_i that $S^A \subseteq \widetilde{pre}[\tau](B_i)$ or $S^A \subseteq \widetilde{pre}[\tau](\neg B_i)$.

At any point of time, one may stop and use the abstractions τ_i^A already computed. One may then either try to prove $Sys^A \models \forall\Box\varphi$ by finite state model-checking. Alternatively one may use the invariant checker tool to apply the method defined by (2) and (3) using the control invariant defined by Sys^A as an auxiliary invariant. In fact, our initial motivation for defining abstract was the construction of such control invariants.

A large number of improvements and variants of this method have been proposed since then. The tools InVest [BL98] and CEGAR [CGJ+00] use spurious counter examples to automatically refine the initial set of predicates. *Invariant checker* implements abstraction refinement and invariant strengthening as described above, but predicate refinement is done manually. A problem with automatic predicate refinement — already proposed in a different setting in [DGG93] — is that it leads easily to an infinite regress, where it is not always the case that a refinement step really improves the abstraction.

The InVest tool proposes also an improvement of the computation of abstract transition relations by trying to directly compute postconditions by quantifier elimination. Invest considers predicate abstractions where individual variables are abstracted by a set of boolean variables. These compositional data abstraction approach is complementary to our control abstractions; and in practice more realistic.

The abstraction and verification tool Bebop/SLAM [BR00, BR01] uses predicate abstraction to extract abstract boolean programs from C programs, and then uses model-checking and counter-example based automatic refinement if needed. This tool has been used with great success to the verification of driver software, where only specific structural properties — such as correct usage of locking mechanisms — are checked.

1.2 Bibliographic References

[ABM93] Y. Afek, G. Brown, and M. Meritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[BL98] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 1998. To appear.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Automatic generation of invariants. In *Conference on Computer Aided Verification CAV'96*, volume 1102 of *LNCS*, 1996.

[BLUP94] Alexander Blinchevsky, Boris Liberman, Ilya Usvyatsky, and Amir Pnueli. Tpvs: Temporal prototype verification system. Progress report, Weizmann Institute, 1994.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, 2001.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. on Computation*, 35 (8), 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[CC80] P. Cousot and R. Cousot. Constructing program invariance proof methods. In *Int. Worksh. on Program Construction, Chateau Bonas*. INRIA, France, 1980.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, LNCS 1855, pages 154–169, 2000.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Symposium on Principles of Programming Languages (POPL 92)*. ACM, January 1992.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[Cri95] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. Symposium on Partial Evaluation and Program Manipulation, La Jolla, California*, June 1995.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proceedings of CAV'93, Crete (GR)*, volume 697, pages 479–490. Lecture Notes in Computer Science, 1993.

[Ger99] R. Gerth. Xxx. *Distributed Computing*, 12, 1999.

[GL93a] S. Graf and C. Loiseaux. Program verification using compositional abstraction. In *TAPSOFT 93, joint conference CAAP/FASE*. LNCS 668, Springer Verlag, April 1993.

[GL93b] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Conference on Computer Aided Verification CAV 93, Heraklion Crete*. LNCS 697, Springer Verlag, 1993.

[GL93c] S. Graf and C. Loiseaux. A tool implementing a method for symbolic program verification. In *Formale Methoden zum Entwurf korrekter Systeme, Bad Herrenalb*, 1993.

[GLW99] S. Graf, Y. Lakhnech, and P. Wolper. Coping with process identities in networks of similar processes. Technical report, Verimag, January 1999.

[GR95] P. Gribomont and D. Rossetto. Caveat: Technique and tool for computer aided verification and transformation. In *Workshop on Computer-Aided Verification (CAV), Liège*. LNCS 939, Springer-Verlag, 1995.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In *Conference on Computer Aided Verification CAV'94, Stanford*. LNCS 818, Springer Verlag, June 1994. a largely improved and extended version appeared in Distributed Computing which is the online version.

[Gra99] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12, 1999. accepted for publication without revision since 1995.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, volume 1102 of *LNCS*, July 1996.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification CAV'97, Haifa*, volume 1254 of *LNCS*, June 1997.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proceedings of Formal Methods in Europe'96*, 1996.

[KDG95] P. Kelb, D. Dams, and R. Gerth. Efficient symbolic model-checking for the full μ-calculus using compositional abstractions. Technical Report Computing Science Reports 95/31, Eindhoven University of Technology, October 1995.

[Kur87] R. P. Kurshan. Reducibilityin analysis of coordination. In *Proc. of Discrete event systems in analysis of coordination*, volume 103 of *Lecture Notes in Computer Science*, pages 19–39, 1987.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design, Vol 6, Iss 1, January 1995*, 1995.

[Loi94] C. Loiseaux. *Vérification symbolique de programmes réactifs à l'aide d'abstractions.* PhD thesis, Université Joseph Fourier, Grenoble, February 1994.

[MBB+95] Zohar Manna, Nikolaj Bjørner, Anca Browne, Edward Y. Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Arjun Kapur, Jaejin Lee, Henny Sipma, and Tomás E. Uribe. Step: The stanford temporal prover. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus*, volume 915 of *Lecture Notes in Computer Science*, pages 793–794, 1995.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. Second Int. Joint Conf. on Artificial Intelligence*, pages 481–489. BCS, 1971.

[PA03] Amir Pnueli and Tamarah Arons. Tlpvs: A pvs-based ltl verification system. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 598–625, 2003.

[Sai98] Hassen Saidi. *Combinaison de Méthodes Déductives et Arithméthiques pour la Vérification.* PhD thesis, Université Joseph Fourier, Grenoble I, January 1998.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. *A Tutorial on Specification and Verification using PVS.* SRI International, Menlo Park, CA, 1993.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large sets of Processes with Network Invariants. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble.* LNCS 407, Springer Verlag, 1989.