Finite Model Checking and Beyond

Bernhard Steffen*
Universität Passau
Gemany

Abstract

This paper reviews finite and infinite-state model checking form two pragmatic perspectives: the application to the static analysis of imperative programs or data flow analysis, and the implementational aspect. Data flow analysis provides an interesting area of application for model checking, which is particularly challenging in the interprocedural case, since infinite state spaces need to be considered. Moreover, as it is a typical case of a structurally unrestricted problem, one should use global iterative methods for solution. This can be exploited for a uniform and efficient treatment based on a *fixpoint analysis machine* that covers all kinds of logics and model structures important for the considered application.

1 Introduction and Motivation

Over the past decade model-checking has emerged as a powerful tool for the automatic analysis of concurrent systems. Whereas model-checking for finite-state systems has already a long history (cf. e.g. [EmLe86, ClES86, Lars88, StWa89, Wins89, BCMD90, Clea90, ClSt91, EnFT91, ClGL92, ClPS93, Lars92, AnSW94]), the theory for infinite systems is still in its initial phase. Bradfield and Stirling [BrSt91] observed that tableaux-based model-checking covers general infinite-state systems. However, their method is not effective. Therefore much work has focussed on *context-free processes*, a subclass of the infinite-state systems. In particular, in [BuSt92] an iterative model-checking algorithm is developed that decides the *alternation-free* part of the modal mu-calculus for *context-free* processes. Moreover, in [HuSt93] it is shown how this can be done using tableaux-based techniques, and in [BuSt94] a model checker for pushdown processes that are a strict generalization of context-free processes has been proposed. More general but also much more expensive are the decision algorithms proposed by Muller and Schupp [MuSc85] for monadic second order logic on pushdown transition graphs implying the decidability of the full mu-calculus [Koze83] for pushdown processes, by Hungar [Hung94] for model checking of macro processes, which are a 'higher order variant' of context-free processes,

*Lehrstuhl für Informatik, Universität Passau, Innstr. 33, D-94032 Passau (Germany), tel: +49 851 509.3090, fax: +49 851 509.3092, steffen@fmi.uni-passau.de

and by Bouajjani et al. who verified non-regular properties for context-free processes [BoER94]. Still, the limits of model checking infinite-state processes are by far not reached, even though it seems to be clear that, except for some few unifying generalizations, practically relevant efficient algorithms will require certain application specific profiles.

In this paper we review (infinite-state) model checking from two pragmatic perspectives, 1) the application to the static analysis of imperative programs (or data flow analysis), which, in the interprocedural case, provides a challenging but still tractable scenario for infinite state model checking, and 2) the implementational aspect, where we propose a uniform platform for efficient implementation.

This choice of application concentrates our attention to *global, iterative* model techniques, as in static analysis properties must be determined *globally*, i.e., for all program points, in order to determine the *full potential* for optimizing program transformations. Moreover, for structurally unrestricted problems of 'tractable' size, standard iterative methods are most efficient. Data flow analysis provides a typical example of this class of problems. Thus, we do not consider local [StWa89, Wins89, Clea90, Lars92, HuSt93, AnSW94] or BBD-based [BCMD90, EnFT91] model checking techniques in this paper, simply because they do not fit the considered application profile.

Concentrating on global iterative model checking techniques allows a *uniform* treatment based on a *fixpoint analysis machine*, which, despite its uniformity, outperforms most of the 'iteration based' special purpose tools. This goal is achieved by translating the specific fixpoint problems into unstructured, very fine-grained but computationally advantageous representations, which allow us to eliminate as much redundancy as possible. Currently we are uniformly covering the kinds of fixpoint computations summarized in Table 1,[1] which are specific to the structure of the underlying model and formula (cf. Section 5). In fact, the differences between these kinds of problems and their corresponding fixpoint computations only require a change in the data domain the fixpoint is computed over and in the program controlling the order of the fixpoint computation. The architecture and all the other data structures of the machine coincide in all these applications.

Fixpoint class	Model class		
	regular	CFR/PDA	macro
homogeneous	(chaotic, $\mathbb{B}$)	(chaotic, $\mathcal{F}_{\mathbb{B}}$)	(chaotic, high.ord. $\mathcal{F}_{\mathbb{B}}$)
hierarchical	(layered, $\mathbb{B}$)	(layered, $\mathcal{F}_{\mathbb{B}}$)	(layered, high.ord. $\mathcal{F}_{\mathbb{B}}$)
alternated nesting	(backtrack, $\mathbb{B}$)	(backtrack, $\mathcal{F}_{\mathbb{B}}$)	(backtrack, high.ord. $\mathcal{F}_{\mathbb{B}}$)

Table 1: Classification of the Fixpoint Applications

The next section sketches the link between data flow analysis and model check-

[1] Here $\mathbb{B}$ denotes the set of Booleans and $\mathcal{F}_{\mathbb{B}}$ the set of Boolean functions.

ing, while Section 3 gives an overview over the scenario addressed in this paper. Subsequently, Section 4 presents the internals of the fixpoint analysis machine. Finally, Section 5 discusses the various forms of fixpoint computations considered, and Section 6 our conclusions and directions to future research.

2 Data Flow Analysis as Model Checking

Data flow analysis (DFA) is concerned with the automatic identification of program points enjoying specific properties, like for example, liveness of variables or equivalence of program terms. Typically, DFA algorithms are constructed for a given program property and therefore have the following functionality:

$$\text{programs} \quad \rightarrow \quad \text{program points enjoying the considered property}$$

Model checking is concerned with the automatic identification of those states of a finite system satisfying a specific modal (or temporal) formula that expresses, for example, properties like deadlock, divergence or liveness. Typically, model checkers are parameterized on the formula of interest and therefore have the following functionality:

$$\text{modal formulas} \times \text{model} \quad \rightarrow \quad \text{states satisfying the argument formula}$$

Identifying programs with models, program points with states and program properties with modal formulas, model checkers can be seen as DFA algorithms that have the program property of interest as a parameter.

This observation can be exploited for the development of an algorithm that automatically generates efficient implementations of DFA-algorithms from specifications written in a modal logic.

In essence, this DFA-*generator* works by *partially evaluating* an appropriate model checker with respect to its modal formula parameter; the result is a standard iterative DFA-algorithm [Hech77], which directly runs on the machine the model checker is implemented on. Using our fixpoint analysis machine, this partial evaluation is part of an optimizing compilation into the corresponding fixpoint analysis machine code.

Currently, our implementation covers standard bit-vector DFA algorithms[2] in an efficient manner: it allows concise high level specifications, and the generated algorithms are guaranteed to be linear in the size of the program being analysed. Moreover it supports the specification of and reasoning about DFA-algorithms. Both can be done structurally by reasoning within the modal logic serving as specification language.

This approach is not limited to intraprocedural analysis via model checking of regular structures, it can also be extended to cover interprocedural data flow analysis via context-free model checking. Having a model checker for context-free structures at hand, this is straightforward for a setting with (recursive) parameterless procedures. However, the clean treatment of parameters is quite

[2] Typical such algorithms determine e.g. live variables, (very) busy expressions or use-definition chains[Hech77].

hard. Surprisingly, the extension to formal procedure parameters, i.e. to higher order, is very complicated but still straightforward [Hung94]. It complicates the domain of the fixpoint computation, leads to an unacceptable runtime complexity, but the framework of specification remains completely unchanged. In contrast, the general treatment of value parameters, which intuitively are much simpler, requires some modification of the whole setting. Fortunately, such modifications are not required for large classes of practically relevant applications [KnRS92, KnRS94, Knoo93].

In [Stef91, Stef93], the framework is illustrated by means of an example of practical relevance: an improved version of Morel/Renvoise's algorithm for eliminating partial redundancies [MoRe79]. The algorithm generated there from a one line specification[3] is as efficient as the standard uni-directional algorithms, because the decomposition of the usual bi-directional structure into a hierarchy of uni-directional analyses guarantees the efficiency of the well-known bit-vector techniques. Moreover, the results of the generated algorithm are provably optimal. In [KnRS92, KnRS94], this optimality result has been strengthened by minimizing the lifetimes of the registers required for the transformation. These modifications could easily be implemented on top of the fixpoint analysis machine.

3 The Scenario

Figure 1 summarizes the scenario discussed in this paper. We will now first explain the top row of the figure addressing the considered application areas, and subsequently sketch the lower part discussing the implementation of the various kinds of model checkers.

- *Intraprocedural dataflow analysis*: algorithms of this kind can be realized efficiently and at almost no implementation cost on our analysis machine via a data flow analysis generator, which automatically produces fixpoint machine code from high level specifications [Stef91, Stef93].
- *Interprocedural data flow analysis*: in this setting we are able to cover a wide class of programs that contain recursive procedures with value parameters. The corresponding data flow analysis generator, which uses the same high level specifications as the intraprocedural version, is under implementation. It requires the combination of the methods presented in [BuSt92, Stef93, BuSt94].
- *Higher order data flow analysis*: this setting allows us to deal with further types of parameters, like reference parameters and procedure parameters. Whereas the case of reference parameters is still rather efficient, the optimal treatment of arbitrary (finite mode) procedures requires an unacceptable effort and should therefore be treated approximatively. The reader is referred to [Knoo93] for details.

[3] Usually, this type of algorithm is given by means of a complex equation system.

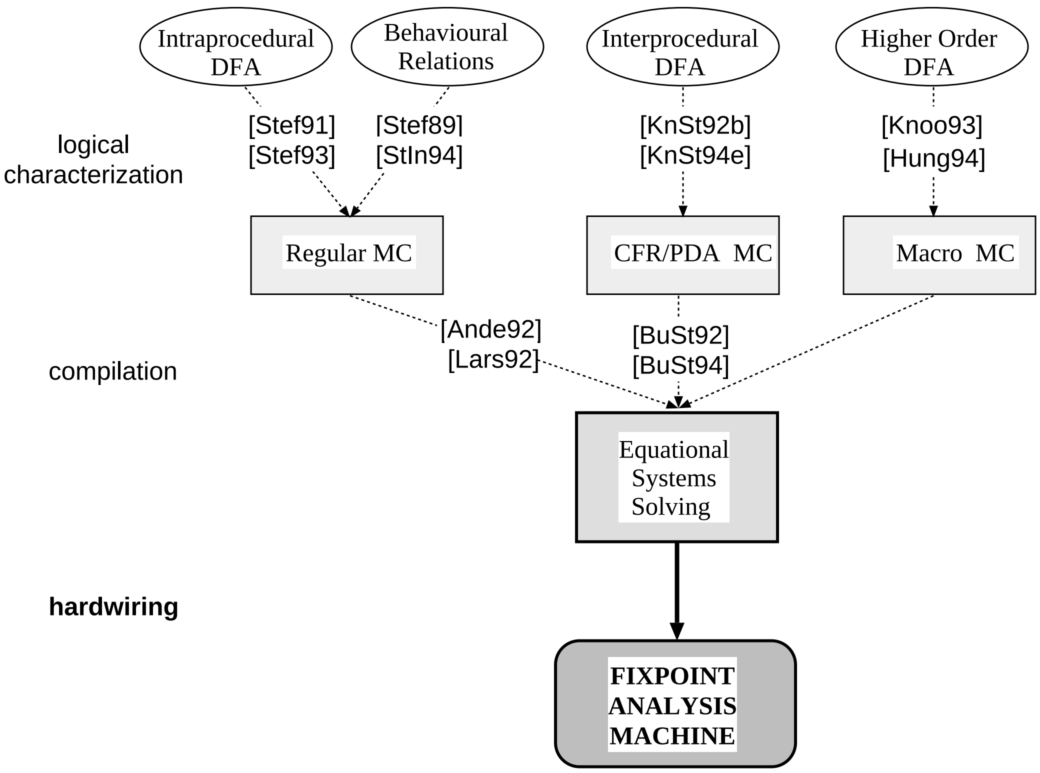

Figure 1: Setup of the Analysis Environment

The remaining top row application concerns the verification of behavioural relations between distributed systems. Like the three applications already mentioned, it can be transformed into a particular model checking problem via logical characterization [Stef89, StIn94].

The second row presents a hierarchy of model checking problems classified according to the structure of the underlying model. Additionally, the possibility and the extent of interference between minimal and maximal fixpoints – homogeneous, hierarchical or alternating fixpoints [EmLe86] – is an important classification criterion. Whereas all the top row applications require at most hierarchical fixpoints, alternating fixpoints are necessary when dealing with properties like fairness. Thus we face a potential of nine structurally different model checking problems, two of which (the alternating case for context-free and macro models, see Table 1), even though decidable, do not yet have a fixpoint characterization. The other seven cases can uniformly be represented by means of a finite equational system together with a parity vector specifying the particular solution of interest, leading to the third row. Actually, we proposed such transformations also for the two exceptional cases, but their correctness is still a conjecture. As most of the translation into the equational representation, which can be regarded as a common intermediate level, is more or less standard, we will concentrate on the analysis machine.

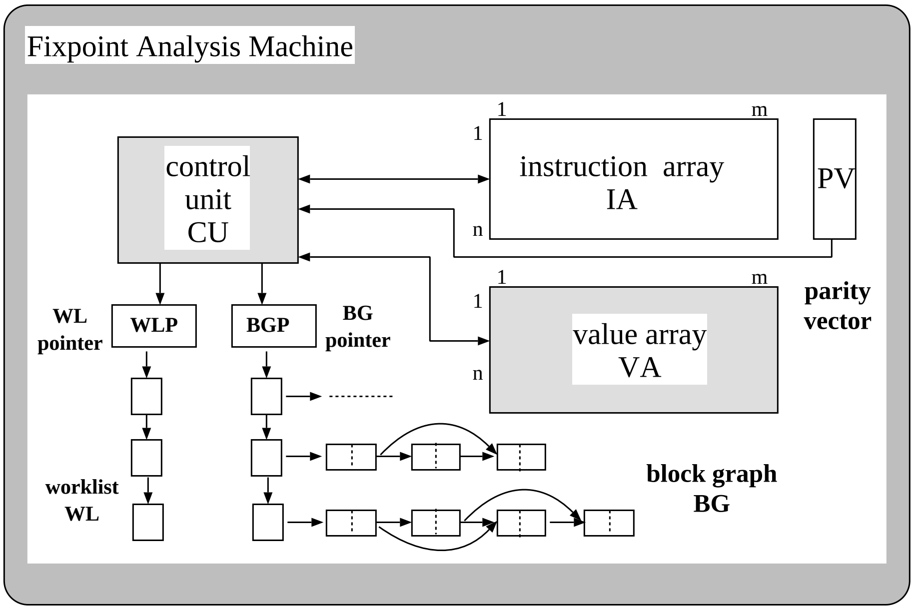

Figure 2: The Architecture of the Fixpoint Analysis Machine.

4 The Fixpoint Analysis Machine

The structure of the analysis machine is a consequence of the observation that, although each of the different kinds of fixpoint computations (cf. Table 1) requires special care, there is a large common core as soon as one breaks the problem down to the appropriate kind of granularity and allows a limited form of parameterization. In fact, the machine architecture we are going to present allows us to uniformly cover all the kinds of fixpoint problems mentioned without performance penalty, as our choice of granularity is tailored for runtime optimization.

The machine architecture is illustrated in Figure 2. Here, the white parts are common to all analyses. Only the value array and the control unit (shaded in the figure) are parameterized in the kind of the problem. The value array is parameterized in the value domain the fixpoint must be computed over, which depends on the kind of model under investigation, and the control unit steers the order of the fixpoint computation according to the considered kind of fixpoint problem.

4.1 The Instruction and Value Arrays

Fixpoint computation proceeds by successively updating the components of the *value array* by means of component-specific operations that are stored in the *instruction array*. Allowing distinct operations for each component of the value array, and being flexible in the choice of the granularity, leads to a very simple structure that must be modeled by the instruction array: *AND/OR/COMP graphs*. They represent (i) the kind of *operation* to perform (conjunction, disjunction, or functional composition), (ii) the *list of operands* for this operation, consisting of components of the value array, as well as (iii) the *influence list*, i.e. those components that are influenced by the value of the considered component. All this can easily and automatically be determined by means of a simple compiler.

Whereas the structure of the instruction array is completely independent of the considered fixpoint problem, the component type of the value array depends on the kind of model we are considering (see Table 1):

- In the *regular* case single bits are sufficient. They indicate whether a specific node of the model satisfies a particular formula.
- In the *CFR/PDA* case the elements correspond to property (predicate) transformers, and are therefore Boolean functions $\mathcal{F}_{\mathbb{B}} : \mathbb{B}^{n \times m} \to \mathbb{B}$ of an arity determined by the size of the considered system and formula. For reasons of efficiency, they are represented as Binary Decision Diagrams (BDDs) [Brya86].
- In the *macro* case, elements correspond to mappings between property transformers. Thus we are dealing with higher order Boolean functions here.

The instruction array is completely constructed during compile-time. Thus there is only read-access at runtime. In contrast, the value array must be read and written at runtime, and only its space allocation is a matter of compilation.

4.2 The Parity Vector

A fixpoint problem is completely specified by a system of mutually recursive equations, where each equation is classified as maximal or minimal according to the kind of fixpoint of interest. The instruction array does not contain this classification. It merely stands for a set of fixpoint problems. We therefore specify the classification separately in a vector, that stores for each row of the instruction array the desired kind of fixpoint, called its *parity*.

Like the instruction array, the parity vector is created at compile-time and only accessed at runtime.

4.3 The Block Graph

Homogeneous fixpoint problems can be solved by means of a totally chaotic iteration over the value array. However, each time both kinds of fixpoints are involved the order of fixpoint computation is essential. For hierarchical systems a layered approach is sufficient, while alternating fixpoint formulas require a very strict discipline involving backtracking. This observation motivates the structure of our block graphs, which are *lists of DAGs*[4] (directed acyclic graphs), where edges represent ordering constraints and nodes collect the set

[4]This choice is an elaboration of the block graphs presented in [ClSt91b, ClKS92]

of all equations which need not be distinguished according to these constraints. This graph structure is already sufficient to uniformly capture even strongly optimized organizations of the fixpoint computation for the alternating case.

Technically, the need for blocks, which are collections of equations whose fixpoints can be computed in an arbitrary order, arises as soon as there are depending minimal and maximal fixpoints. This dependency requires a strict organization of the order in which these fixpoints are computed. In the hierarchical case, a simple sequentialization is required, and in the more complicated alternating case a backtracking procedure must be organized following the strucure of the underlying block graph. It is convenient to additionally split blocks according to their parity, which leads to the notions of min-blocks and max-blocks. This additional separation is uncritical as a single block could anyhow only comprise completely independent minimal and maximal equations.

Blocks are important, as they allow an efficient fixpoint computation. The switching between different blocks according to ordering constraints is more expensive then the worklist-oriented chaotic iteration allowed within a block. Thus efficient fixpoint algorithms will completely determine the fixpoint of a block before taking a switch. This approach is only guaranteed to be optimal when using counters [ClSt91, ClSt91b].

- For *homogeneous fixpoints*, a one component list containing a one element DAG is sufficient, as we are dealing with a single block here where no ordering constraints need to be taken care of.
- *hierarchical fixpoint* computations postpone the evaluation of an equation until all equations of different parity which may influence it have reached their fixpoints. In terms of model checking, this corresponds to an 'inner-most' strategy for the evaluation of the formula. This requirement can be expressed sufficiently by means of a list of sets of formulas, i.e. by a list of one component DAGs.
- *alternating fixpoint* computations require backtracking, which leads to an exponential complexity in the alternation depth ([EmLe86]) of the considered formula. Thus the most important source of optimization is the reduction of backtracking steps. Block graphs support such a reduction by structuring the global dependence graph between the fixpoint equations in the following fashion:
  - The list structure reflects the dependence (ordering constraints) between the strongly connected components of the dependence graph. Of course, these constraints form in general a DAG structure. However, as in the hierarchical case, we can simply collapse this DAG to a list without runtime penalty.
  - The DAG structure reflects part of the ordering constraints within a strongly connected component: a constraint between two equations e_1 and e_2 is only kept if the row of e_1 precedes the row of e_2 in the value array. This DAG of equations is then collapsed by combining all equations that have a 'similar' dependence relation into a node.

Block graphs are constructed at compile-time, and there is only read-access at runtime.

4.4 The Worklist

Whereas the block graph is a mean to steer the fixpoint computation globally, i.e. between blocks, the worklist organizes the fixpoint computation inside a block. It contains the addresses of the elements of the current block whose values must be updated as a consequence of earlier changes in the value array. The list is dynamically initialized when entering a new block, and it is updated during the computation by appending the addresses of all the influenced value array components. The worklist is a pure runtime entity, initialized, updated, and read at runtime.

5 Computing Fixpoints

In this section we sketch the computation mechanism for the various kinds of fixpoint problems. We start by considering the three regular problems, which only require a fixpoint computation for bitvectors. Subsequently we discuss the extension to context-free structures. Even though the decidability of the model checking problem is implied by decidability results about monadic second order logic [MuSc85], the known efficient algorithms cover alternation free, i.e. hierarchical formulas, only. The best known algorithm for the general case is non-elementary. The further extension to push-down structures, which in contrast to classical automata theory do not coincide with context-free structures when the branching structure of the models are essential, is rather straightforward and still in the range of tractability. This is no longer the case for macro structures [Hung94], which require a very expensive higher-order treatment and will not be discussed here.

5.1 First-Order Fixpoints

Uniform fixpoints: Homogeneous and regular problems can be regarded as the common core of all the regular fixpoint computations. It consists of the determination of the fixpoint over a single block. This computation is performed by successively decrementing counters until the fixpoint is reached, a process which is steered by a worklist that contains references to all the components whose counters are currently known to require decrement. The worklist is updated by adding references to all influenced components whenever one counter becomes zero, which indicates a change of its corresponding Boolean value [ClSt91, ClSt91b].

Hierarchical Fixpoints: Here, blocks are sequentially computed in the order indicated by the block graph. The fixpoint computation within the blocks is identical to the one in the homogeneous case. It should be noted that the counters for a block must be constructed immediately before its fixpoint computation. This is necessary in order to capture the effects of the earlier fixpoint computations.

Alternating Fixpoints: Again blocks are treated exactly as in the homogeneous case. However, in contrast to the previous two cases, this computation must be repeated according to changes in blocks of different parity that are higher in the hierarchy but still in the same strongly connected component.

5.2 Second-Order Fixpoints

Structurally, these fixpoint computations follow exactly the same lines as in the first order case. Only the domain of the components of the value array is now *second order*, i.e., instead of determining properties for states of the considered systems, we must determine property transformers for certain classes of states. Moreover, in the alternating case it is still an open problem whether the straightforward extension to the second order domains computes the intended values. We hope that experimenting with our machine will help us clarifying this point.

We are now going to sketch the essence of an algorithm that decides the alternation-free modal mu-calculus for *context-free* processes, i.e. for processes that are given in terms of a context-free grammar, or equivalently, as mutually recursive systems of finite state-labelled transition systems. For the understanding of the key idea behind the model checking algorithm, the second perspective is advantageous, as it stresses the transformational character of the approach: processes are represented as sets of named standard finite-state labelled transition systems, where the label set additionally contains the names identifying the different transition systems. A transition labelled with such a name should be viewed as a call to the 'procedure' (transition system) identified by this name.[5] Operationally, these possibly recursive procedure calls are treated by *recursive action refinements*, i.e. by the copying process illustrated in Figure 3 for a single recursive named transition system CM.

CM defines a little encoding machine, which reads a word over the alphabet $\{a, b\}$ until it finds the word delimiter #. Then, in its second phase, it outputs the corresponding sequence of codes in reverse order.

The right hand side shows CM after a single expansion step, whose infinite repetition yields a unique infinite transition system containing no name-labelled transition.[6] It is exactly this infinite transition system, which is represented by CM, and which, as we will argue below, can be model checked by applying a second order model checker to its finite representation CM.

The heart of our algorithm is to iteratively compute for each node σ of the finite process representation the *property transformer* telling which properties are valid at this node depending on the properties considered to be valid after

[5] Considering names as nonterminals and the other labels as terminals establishes the match to context-free processes.

[6] Technically, we are restricting our attention to representations that are guarded, i.e. whose initial transitions are not labelled by names, and which have a unique start and end node.

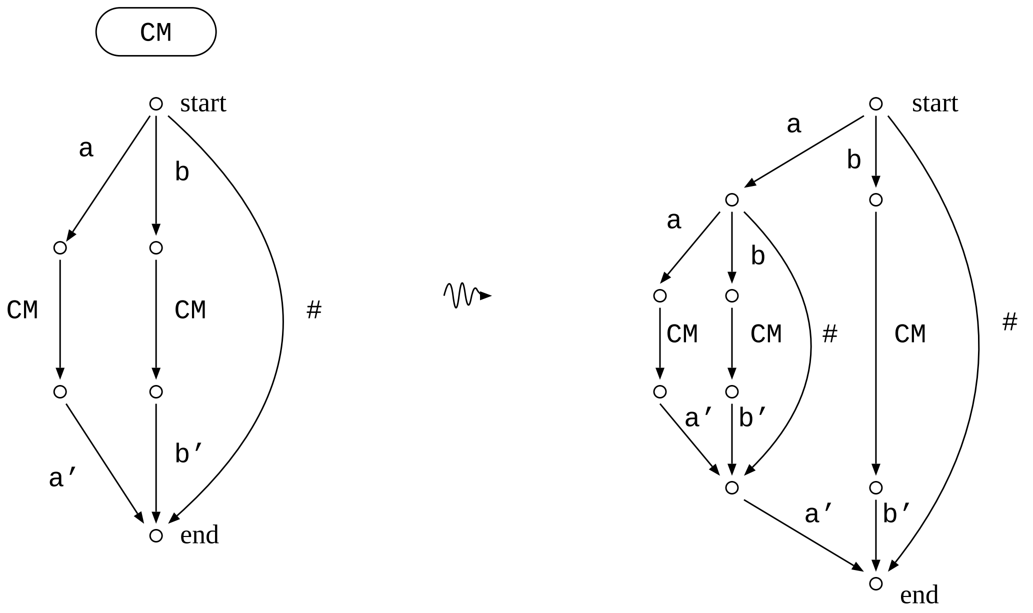

Figure 3: Expanding a Procedural Transition System

the 'termination' of the procedure containing σ. The subsequent decision step is more or less trivial. This results in the following two step structure:

1. iterative computation of a *property transformer* for each state class of the finite process representation. This computation works block-wise. In the case of a *min*-block it successively updates the initial "minimal" predicate transformers until the smallest fixpoint is reached. The handling of *max*-blocks is completely dual, and the global algorithm, which deals with a hierarchy of blocks proceeds hierarchically as described in [ClSt91, ClKS92].

   The iteration mechanism as such is essentially the same as for usual iterative algorithms, except for:

   - the handling of the procedural identifiers (non-terminals). They are dealt with by applying the currently valid approximation for their defining procedural transition systems.
   - the domain for the iteration, which is second order here, i.e. it consists of property transformers (Section 3).

   This computation is effective because we can restrict our attention to the finitely many subformulas of the property currently being investigated.

2. decision of the model-checking problem: having the predicate transformers at hand, the model-checking problem can be solved simply by checking whether the formula under consideration is a member of the set of formulas that results from applying the property transformer of the "main

procedure" to the set of formulas that are valid for a deadlocking or terminating state.

The complexity of the resulting algorithm is linear in the size of the system's representation and exponential in the size of the property. This is quite promising, as many practically relevant problems can be composed of very small properties: e.g. bitvector analyses, which are common in data flow analysis have exponent one ([KnSt93a])!

6 Concluding Remarks

We have reviewed finite and infinite-state model checking from two pragmatic perspectives: the application to the static analysis of imperative programs or data flow analysis, and the implementational aspect. This application driven review gives some further evidence for the general observation that practically relevant efficient verification algorithms will require certain application specific profiles.

Of course this observation is not new and there is already a wide range of application or structure specific approaches to fight the well-known state explosion problem, ranging from the most elegant and ambitious *compositional* methods [AnSW94, ClLM89, GrSt90], over *partial order* methods that suppress unnecessary interleavings of actions [GoWo91, GoPi91, Valm93] to *Binary Decision Diagram*-based codings that work for an interesting class of practically relevant systems [Brya86, BCMD90, EnFT91]. All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be dramatically reduced by suppressing details that are irrelevant for verification [CoCo77, ClGL92, ClRi93, DaGG93, GrLo93]. This list of references and approaches is by far not exhaustive and still increasing. Thus we are faced with an almost unmanageable variety of approaches and algorithms asking for meta-tools managing this diversity. The DaCapo environment applied to the organization of UNIX commands in [SFCM94] addresses this need.

Acknowledgement

The work presented in this paper was strongly influenced by collaboration and intensive discussion with Olaf Burkart, Andreas Claßen, Marion Klein, Jens Knoop, Gerald Lüttgen, Tiziana Margaria, and Oliver Rüthing.

References

[AhSU85] A. Aho, R. Sethi, J. Ullman: "*Compilers: Principles, Techniques, and Tools*," Addison Wesley, 1985.

[AnSW94] H. Andersen, C. Stirling, G. Winskel: "*A Compositional Proof System for the Modal Mu-Calculus*," Proceedings 5th IEEE LICS, 1994.

[BCMD90] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang: "*Symbolic Model Checking:* 10^{20} *States and Beyond*," Proc. of 5th IEEE LICS, pp. 428–439, Philadelphia, 1990.

[BoER94] A. Bouajjani, R. Echaed, R. Robbana: "*Verification of nonregular temporal properties for context-free processes*," Proc. CONCUR'94, Uppsala (Sweden), Aug. 1994, LNCS N.836, pp.81-97, Springer.

[BrSt91] J. Bradfield, C. Stirling: "*Local Model Checking for Infinite State Spaces*," Tech. Report ECS-LFCS-90-115, LFCS, Edinburgh, 1991.

[Brya86] R. Bryant: "*Graph-Based Algorithm for Boolean Function Manipulation*," IEEE Trans.on Computers, Vol. C-35, No. 8, pp. 677-691, 1986.

[BuSt92] O. Burkart, B. Steffen: "*Model Checking for Context–Free Processes*," Proc. of CONCUR '92, Stony Brook (NJ), August 1992, LNCS 630, pp. 123–137, Springer Verlag.

[BuSt94] O. Burkart, B. Steffen: "*Pushdown Processes: Parallel Composition and Model Checking*," Proc. of CONCUR '94, Stockholm (Sweden), August 1994, LNCS N. 836, pp. 98-113, Springer Verlag.

[Clea90] R. Cleaveland: "*Tableau-Based Model Checking in the Propositional Mu-Calculus*," Acta Informatica N. 27, pp.725–747, 1990.

[ClES86] E. Clarke, A. Emerson, A. Sistla: "*Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*," ACM TOPLAS, V.8 N.2, pp. 244–263, 1986.

[ClGL92] E. Clarke, O. Grumberg, D. Long: "*Model Checking and Abstraction*," Proceedings POPL'92.

[ClKS92] R. Cleaveland, M. Klein, B. Steffen: "*Faster Model Checking for the Modal Mu-Calculus*," Proc. of CAV '92, Montreal (Canada) LNCS 663, pp. 410–422, Springer Verlag, 1992.

[ClLM89] E. Clarke, D. Long, K. McMillan: "*Compositional Model Checking*," Proceedings LICS'89.

[ClPS93] R. Cleaveland, J. G. Parrow, B. Steffen: "*The concurrency workbench: A semantics-based verification tool for finite state systems*," ACM TOPLAS, Jan. 1993.

[ClRi93] R. Cleaveland, J. Riely: "*Testing-Based Abstractions for Value Passing Systems*," Proc. of CONCUR '94, Stockholm (Sweden), August 1994, LNCS N. 836, Springer Verlag.

[ClSt91] R. Cleaveland, B.Steffen: '*A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus*," Proc. CAV '91, Aalborg (Denmark), July 1991, LNCS N.575, pp.48–58, Springer.

[ClSt91b] R. Cleaveland, B.Steffen: "*Computing Behavioural Relations, Logically,*" Proc. ICALP'91, Segovia (Spain), Aug. 1991, LNCS N.510, Springer.

[CoCo77] P. Cousot, R. Cousot: "*Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction and Approximation of Fixpoints,*" Proceedings POPL'77.

[DaGG93] D. Dams, O. Grumberg, R. Gerth: "*Generation of Reduced Models for Checking Fragments of CTL,*" Proc. CAV'93, Crete (GR), June 1993, LNCS 697, Springer Verlag, 1993, pp.479–490.

[EnFT91] R. Enders, T. Filkorn, D. Taubner: "*Generating BDDs for Symbolic Model Checking in CCS,*" Proc. CAV '92, Aalborg (Denmark), July 1991, LNCS N.575, Springer Verlag.

[EmLe86] A. Emerson, C.-L. Lei: "*Efficient Model Checking in Fragments of the Propositional Mu-Calculus,*" Proc. of LICS'86, IEEE Computer Society Press, pp. 267–278, 1986.

[GoPi91] P. Godefroid, D. Pirottin: "*Refining Dependencies Improves Partial-Order Verification Methods,*" Proc.CAV'93, Elounda (Greece) June/July 1993, LNCS N.697, pp.397–408.

[GoWo91] P.Godefroid, P.Wolper: "*Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties,*" Proc. CAV'91, Aalborg (Denmark), Juli 1991, LNCS N.575, 1991, pp.332–342.

[GrLo93] S. Graf, C. Loiseaux: "*Program Verification using Compositional Abstraction,*" Proceedings FASE/TAPSOFT'93.

[GrSt90] S. Graf, B. Steffen: "*Using Interface Specifications for Compositional Minimization of Finite State Systems,*" Proceedings CAV'90, Rutgers (NJ), June 1990, LNCS N. 407, Springer Verlag, 1991.

[Hech77] M. Hecht: "*Flow analysis of computer programs,*" Elsevier, North-Holland, 1977.

[Hung94] H. Hungar: "*Model Checking of macro processes,*" Proc. of CAV'94, Palo Alto (CA), June 1994, LNCS N.818, Springer V., pp.169–181.

[HuSt93] H. Hungar, B. Steffen: "*Local Model–Checking for Context–Free Processes,*" Proc. ICALP'93, LNCS N. 700, Springer Verlag, pp. 593-605. Accepted for the *Nordic Journal of Computer Science*, 1993.

[Knoo93] J. Knoop: "*Optimal interprocedural program optimization: A new framework and its application,*" PhD thesis, Dep. of Computer Science, Univ. of Kiel, Germany, 1993. To appear as LNCS monograph, Springer V.

[KnRS92] J. Knoop, O. Rüthing, B. Steffen: "*Lazy Code Motion*," Proc. PLDI Conference'92, San Francisco, CA, June 1992, ACM SIGPLAN Notices, Vol.27, pp. 224–234.

[KnRS94] J. Knoop, O. Rüthing, B. Steffen: "*Optimal Code Motion: Theory and Practice*," ACM Transactions on Programming Languages and Systems, Vol. 16, N.4 (1994), pp.1117 - 1155.

[KnRS94e] J. Knoop, O. Rüthing, B. Steffen: "*A tool kit for constructing optimal interprocedural data flow analyses*," Fakultät für Mathematik und Informatik, Univ. Passau, Germany, MIP-Bericht Nr. 9413 (1994).

[KnSt92b] J. Knoop, B. Steffen: "*The interprocedural coincidence theorem*," Proc. CC'92, Paderborn (Germany), Springer-Verlag, LNCS 641, pp.125 - 140.

[KnSt93a] J. Knoop, B. Steffen: "*Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework*," Inst. für Informatik und Praktische Mathematik, Universität Kiel (Germany), Bericht Nr. 9309 (1993).

[Koze83] D. Kozen: "*Results on the Propositional μ-Calculus*," TCS Vol. 27, pp. 333–354, 1983.

[Lars88] K.G. Larsen: "*Proof Systems for Hennessy–Milner Logic with Recursion*," Proc. CAAP '88, LNCS N. 299, pp.215–230, Springer V.

[Lars92] K.G. Larsen: "*Efficient Local Correctness Checking*," Proc. of CAV'92, Montreal (CAN), LNCS N.663, pp.410-422, Springer V.

[MoRe79] E. Morel, C. Renvoise: "*Global optimization by supression of partial redundancies*," Comm. of the ACM N.22, Vol. 2, pp. 96–103, 1979.

[MuSc85] D. Muller, P.Schupp: "*The Theory of Ends, Pushdown Automata, and Second-Order Logic*," TCS N. 37, pp. 51–75, 1985.

[SFCM94] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*Intelligent Software synthesis in the* DaCapo *Environment*," Proc. 6th Nordic Workshop on Programming Theory, Aarhus (Denmark), October 1994, same volume.

[Stef89] B. Steffen: "*Characteristic Formulae*," Proc. of ICALP'89, Stresa (Italy), LNCS N.372, Springer Verlag, 1989.

[Stef91] B. Steffen: "*Data Flow Analysis as Model Checking*," Proc. TACS'91, Sendai (Japan), LNCS N.526, pp. 346–364, Springer V., 1991.

[Stef93] B. Steffen: "*Generating Data Flow Analysis Algorithms from Modal Specifications*," Science of Computer Programming N.21, 1993, pp.115-139.

[StIn94] B. Steffen, A. Ingólfsdóttir: "*Characteristic Formulae for Finite State Processes*," Information and Computation, Vol.110, No.1, 1994.

[StWa89] C. Stirling, D. Walker: "*Local Model Checking in the Modal Mu-Calculus*," Proc. TAPSOFT '89, LNCS N.351, pp.369–383, Springer.

[Valm93] A. Valmari: "*On-The-Fly Verification with Stubborn Sets*," Proc.CAV'93, Elounda (GR) June 1993, LNCS N.697, pp.397–408.

[Wins89] G. Winskel: "*A Note on Model Checking the Modal Mu-Calculus*," Proc. ICALP '89, Stresa (Italy), LNCS N.372, pp. 761–772, Springer.