Published in Proc. CONCUR'95, Int. Conference on Concurrency Theory, LNCS, volume 962. Springer Verlag, The Fixpoint-Analysis Machine

Bernhard Steffen^{*}

Andreas Claßen Marion Klein Jens Knoop

Universität Passau

Tiziana Margaria

Germany

Abstract. We present a fixpoint-analysis machine, for the efficient computation of homogeneous, hierarchical, and alternating fixpoints over reqular, context-free/push-down and macro models. Applications of such fixpoint computations include intra- and interprocedural data flow analysis, model checking for various temporal logics, and the verification of behavioural relations between distributed systems. The fixpoint-analysis machine identifies an adequate (parameterized) level for a uniform treatment of all those problems, which, despite its uniformity, outperforms the 'standard iteration based' special purpose tools usually by factors around 10, even if the additional compilation time is taken into account.

Introduction and Motivation 1

A great number of analysis and verification problems such as abstract interpretation, data flow analysis, model checking, determination of behavioural relations between distributed systems, hardware verification and synthesis, etc., boil down to the computation of a specific kind of fixpoint. In fact, in all these areas of application, specific fixpoint solving tools have been independently constructed and tuned for their special purposes.

The idea behind the fixpoint-analysis machine is to define a uniform platform for fixpoint computations, which, despite its uniformity, outperforms the 'iteration based' special purpose tools. This goal is approached by translating the specific fixpoint problems into very fine-grained but computationally advantageous representations, which allow us to eliminate as much redundancy as possible. Our design stems from the observation that almost all fixpoint problems considered in practice can be formulated as a model checking problem of a certain kind. Currently we are uniformly covering the kinds of fixpoint computations summarized in Table 1, which depend on the structure of the underlying model and formula. The structure of the analysis machine is a consequence of the observation that, although each of the different kinds of fixpoint computations requires special care, there is a large common core as soon as one breaks down

Lehrstuhlfür Programmiersysteme, Universität Passau, Innstraße 33. D-94032 Passau (Germany), tel: +49 851 509.3090, fax: +49 851 509.3092, steffen@fmi.uni-passau.de

the problem 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. 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. Thus the architecture and the remaining data structures of the machine coincide in all these applications.

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

Table 1. Classification of the Fixpoint Computations as (strategy, domain)

Our practical experience with the analysis machine confirms the well-known fact that compilation is better than interpretation. In fact, even taking the additional translation effort into account, the analysis machine outperforms the computations on standard data structures by factors usually around 10. Details are reported in Section 6.

The uniform and general structure of our machine also leads to excellent experimental features. On the practical side, it supports the investigation of heuristics, which are important when dealing with complex kinds of fixpoint problems. On the theoretical side, it supports the study e.g. of the essence of alternated nesting, which is still a matter of research.

It should be noted that we focus on global iterative fixpoint computations here. Therefore very specific heuristics (like e.g. the Binary Decision Diagrambased techniques for model checking) which are extremely efficient in 'good' cases, but much worse than the standard iteration techniques in 'bad' cases, are not covered.

The fixpoint machine constitutes a central component of the computational core of the META-Frame [StMC95, SFCM94, MaCS95], which is a uniform environment for high-level construction, verification and analysis of hardware and software systems.

The next section summarizes the domain of application, while Section 3 presents our analysis machine. Subsequently, Section 4 discusses the optimizing compilation, Section 5 describes the fixpoint computation mechanism, Section 6 reports on the performance of the machine, and Section 7 draws our conclusions.

2 The Application Domains: The Present Scenario

Figure 1 summarizes the scenario discussed in this paper. The upper two rows of the figure address the currently considered application areas, ranging from several kinds of dataflow analyses [Hech77] and the verification of behavioural relations (top row) to various classes of model checking (second upper row). As each of the top row applications can be reduced to model checking via logical characterization without runtime penalty, the second upper row, showing a hierarchy of model checking problems, represents both an application level and a common platform for the top row applications.

We will first explain the upper two rows of the figure, 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, Stef94]. We will discuss this implementation and its performance in Section 6.



Fig. 1. Setup of the Analysis Environment

- 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 [Stef93, BuSt94, KnSt92a, KnRS94].
- Higher order data flow analysis: this setting allows us to deal with further types of parameters, like reference 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 handled approximatively. For details the reader is referred to [Knoo93], where a method is presented that is safe, efficient, and optimal for programs of mode two without global formal procedure parameters. Essentially, all these methods boil down to adding a specific preprocess to an interprocedural analysis, which can efficiently be realized on the analysis machine. The corresponding extension of the interprocedural data flow analysis generator to this case is rather simple, as the preprocess is independent of the specific analysis.
- Behavioural relation checking: the verification of behavioural relations between distributed systems can also be transformed into a particular model checking problem via logical characterization [Stef89, StIn94].

The second row of Figure 1 presents a hierarchy of model checking problems that are classified according to the structure of the underlying model [Stef94]. Beside this structure, the possibility and the extent of interference between minimal and maximal fixpoints – generating 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. Altogether we face a potential of nine structurally different model checking problems, two of which (the alternating case for context-free/push-down and macro models), see Table 1, even though decidable, do not have a fixpoint characterization yet. The other seven cases can be uniformly represented by means of a finite equational system together with a parity vector specifying the particular solution (minimal, maximal) of interest. This leads to the third row. Actually, we proposed such transformations also for the two exceptional cases, but their correctness is still a conjecture. While standard model checkers work on the representation level of equational systems, our computation takes place on the level of the fixpoint-analysis machine. Moreover, 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 in the sequel.

Besides presenting the architecture of the analysis machine, the next sections will show how to translate an equational description of the third level into machine code. In fact, the point of this paper lies in the impact of this optimizing translation on the performances, which will be discussed in detail for the hierarchical regular setting. The extensions to other settings are more complicated, not yet fully implemented, and will therefore only be sketched.

3 The Fixpoint-Analysis Machine

The structure of the machine reflects the observation that, although each of the different kinds of fixpoint computations of Table 1 requires special care, there is a large common core as soon as one breaks down the problem 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 mentioned kinds of fixpoint problems 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 problem. The parameter for the value array is the type of values considered (the second component in Table 1), which depends on the kind of model under investigation, and the control unit steers the order (chaotic, layered or with backtrack) of the fixpoint computation accordingly. Instruction array, parity vector, block graph, and worklist are completely problem-independent. The following paragraphs sketch the 'abstract data type' of each functional unit.



Fig. 2. The Architecture of the Fixpoint-Analysis Machine.

3.1 The Instruction and Value Arrays

The fixpoint computation proceeds by successively updating the components of the value array (encoding the information of n equations for m states of the model) by means of component-specific operations that are stored in the *instruction array*. Our choice of 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, in form of a list of components of the value array, and (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 (see section 4).

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, second component):

- 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 $\mathcal{F}_{\mathbf{B}} : \mathbb{B}^n \to \mathbb{B}$ 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 at 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.

3.2 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. The classification is specified separately in a vector that stores for each row of the instruction array the desired kind (min, max) of fixpoint, called its *parity*. Note that it would be technically simple, but algorithmically unpleasant, to allow changes of the parity component-wise, but there is currently no demand in this direction.

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

3.3 The Block Graph

Homogeneous fixpoint problems can be solved by means of a totally *chaotic* iteration [GKLR94] over the value array. But whenever both kinds of fixpoints

are involved, the order of the fixpoint computation becomes 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* $DAGs^2$ (directed acyclic graphs). The basic underlying idea is that edges represent ordering constraints and nodes collect *blocks*, i.e., collections of equations whose fixpoints can be computed in an arbitrary order. In fact, we will see that this graph structure is already sufficient to uniformly capture even the strongly optimized organization of the fixpoint computation for the alternating case.

Technically, the need for blocks 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 (layered computation) is sufficient, 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 *maxblocks*. 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 than 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 (cf. Section 5).

- For homogeneous fixpoints, a one component list containing a single node DAG is sufficient, as we are here dealing with a single block 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 'innermost' 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 node 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.

² This choice is an elaboration of the block graphs presented in [ClSt91b, ClKS92].

• 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. We will here omit the exact definition of this rather complicated collapse.

Block graphs are constructed at compile-time, and there is only read-access at runtime. In fact, we only need the operations **NEXT_BL** to access the next set of equations, whose evaluation can be performed in an arbitrary order in case no backtracking is required, and **RESET_BL**, to provide a similar set in case backtracking is needed.

3.4 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 value array components 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.

4 Optimizing Compilation

The organization of the impact of the interference between minimal and maximal fixpoints on the fixpoint computation by means of the block graph is an essential part of the compilation. As this has been discussed already, and as the corresponding programs of the control unit are rather straightforward, we concentrate here on the treatment of the different kinds of models (regular, CFR/PDA, macro). As mentioned already, this only concerns the value array, even though the instruction array is indirectly affected too, since simple data domains support more optimizations. In particular, we will see that the partial evaluation feature of our compiler completely evaluates all function compositions in the regular case.

A central feature of the translation is *partial evaluation*. Whereas certain basic techniques are always applied, more specific techiques are used depending on the analysis context. We explain this in the context of a model checking problem starting with the standard case.

1. The knowledge of the *logic formula* alone suffices to determine the kind of fixpoint to be computed, the involved AND/OR/COMP subformulas, their parity, and part of their organization in blocks. This instantiation provides a tool for checking the considered formula for arbitrary models. Knowing the kind of models to be considered allows us to determine the domain of the

fixpoint computation. Machines of this kind correspond to the usual intraor interprocedural data flow analysis algorithms ([Stef91, Stef93]). Beside this straightforward partial evaluation, our compilers also contain a rewriting machine, which aims at a minimal equational characterization of

- the considered formula. This rewriting machine is rather complex, thus it should only be applied if the formula will be used for the investigation of several models, as it is the case in data flow analysis.
- 2. The knowledge of the *model* under investigation determines the domain of the fixpoint computation and preliminary versions of the influence and dependence sets are fixed.

Beside this partial evaluation, we also provide minimization procedures that e.g. collapse the model up to bisimulation ([Miln89]). This step only makes sense if a single model is going to be investigated with respect to several formulas. Typically, this arises during the development of a system, when designers want to verify certain safety and liveness properties for their design.

The results of the separate compilation steps above are then merged to a single combined representation, which is the basis for the instantiation of the instruction array. Of course, if both the model and the formula are known already at the beginning, the instruction array is directly instantiated.

Finally, we discuss how the AND/OR/COMP functions constituting the components of the instruction array can be further optimized:

- 1. Functional composition is necessary to describe the effect of a transition step in the model. If the effect of this step is known at compile-time, which is e.g. the case when modelling intraprocedural (i.e. regular) analyses, all the functional compositions can be immediately evaluated. In the more complicated case of interprocedural (i.e. CFR/PDA) analyses, some of the transitions denote procedure calls. Thus their effect is not known at compile-time. However, the functional compositions associated with all the other transitions can still be evaluated. The runtime gain of this partial evaluation is usually much higher than the partial evaluation time itself.
- 2. Several entries of the instruction array will be constant functions. Thus we can perform constant propagation and folding on the instruction array.
- 3. A particularly strong optimization is possible for regular model structures, i.e. in cases where the components of the value array store bits. From the first step we know already that we only need to consider conjunction and disjunction in this case. This observation leads to the introduction of counters, which intuitively measure the distance to a change in the value array. We will explain this idea, a modified version of which can already be found in [ClSt91], in the case of a homogeneous system of maximal equations. Homogeneous systems of minimal equations behave dually.

Maximal fixpoints are computed by successively updating a maximally initialized value array, where (essentially) all components are assumed to be true [Tars55]. Thus for monotonicity reasons, an update can only switch from true to false. For a conjunction, this happens as soon as one operand switches

to *false*. Thus the counter is initialized to **1**. In contrast, for disjunction all the operands must switch before the value changes. Thus the counter is initialized with the number of operands. Working on counters avoids to evaluate any of the instructions of the instruction array, as the only operation we need is a decrement of the corresponding counter whenever one of the operands changes its value. Only when the counter of an array component reaches zero (indicating a switch of its corresponding boolean value) all influenced components need to be informed to decrement their counters via insertion in the worklist.

This optimization is also applicable to the fixpoint computation for single blocks in the hierarchical and alternating case.

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 is 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

Homogeneous Fixpoints: The case of homogeneous 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. As indicated in the previous section, this computation is performed on a counter array, by successively decrementing counters until the fixpoint is reached. This process 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 became 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 initialized immediately before its fixpoint computation. This is necessary in order to capture the effects of the earlier fixpoint computations [ClSt91b].

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. A detailed description of this procedure is rather complicated (cf. [ClSt91b, ClKS92]), and omitted here.

5.2 Second-Order (and Higher Order) Fixpoints

Structurally, these fixpoint computations follow exactly the same lines as the first order case. Only the domain of the value array components 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, with the consequence that the 'counter optimization' is not applicable. 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.

The essence of our algorithm deciding the alternation-free modal mu-calculus for *context-free* processes, i.e. for processes that are given in terms of a contextfree grammar, becomes apparent when viewing these processes as mutually recursive systems of regular processes. In this case, the regular component processes contain *call transitions* that are labelled with the name of the called component process.³ Our algorithm works directly on this 'procedural' process representation. Its heart is the computation of *property transformers* telling which properties (formulas) are valid at the nodes of a component process depending on the properties considered to be valid after the 'termination' of this component process. The subsequent decision step completing the model checking procedure is straightforward. See [BuSt92, BuSt94] for details.

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 Implementation and Performances

A prototype of the fixpoint-analysis machine has been implemented in C++ as part of the META-Frame([StMC95, SFCM94, MaCS95]), our environment for the development of heterogeneous analysis and verification tools, which currently runs on a SUN SparcStation 20 under UNIX. In order to give an impression of the performance gain, we report on two series of examples.

The first series deals with the verification of hierarchical properties of increasing size for versions of Milner's scheduler with growing numbers of agents

³ Considering the labels of these call transitions as nonterminals and the other labels as terminals establishes the formal match to context-free processes.



Fig. 3. Scheduler Performance Gain.

[Miln89]. The schedulers are represented by regular models and the investigated properties are checked by a hierarchical fixpoint analysis. To get an indication to the impact of the property size we checked an increasing number of conjunctions of a basic property expressing aspects of the alternating behaviour of the cells. Figure 3 graphically demonstrates the performance gain when comparing the runtime results of our FAM analysis machine with the 'conventional' model checker CMC presented in [ClKS92]. Initially the gain rises very quickly for properties of increasing size before it asymptotically approximates constant factors between 10 and 18 for the schedulers with 5, 7, and 9 agents.

Figure 2 shows the individual runtime results for the various properties and schedulers. For the scheduler with 5 agents we detail the total time as the sum of the time for the partial evaluation and configuration (config.), and the analysis. One observes that the higher the analysis share of the total runtime, the higher the performance gain of the analysis machine, i.e. the initial partial evaluation and configuration of the machine are better exploited if the considered problems are hard in the sense of requiring a high computation effort compared with the initialization phase. Furthermore, the higher branching factors of the larger schedulers also favour the analysis machine that determines the models' successor information only once in the partial evaluation step prior to the actual fixpoint analysis.

The second series of examples checks a property of alternation depth 2 with different parameters of the modal operators for a sequence of regular models M_k of increasing size (cf. Figure 4). The modal property expresses that the atomic proposition A holds infinitely often along all $(\{a, b, c\})$ paths. Assuming that all transitions are labeled with $(\{a, b, c\})$, this is only true for state v as all other states reach the loop at s which does not satisfy A. For model M_k with k states

N. of Property Conjuncts		1	2	4	8	16	32	64	128	
Scheduler 5	CMC	config.	0.03	0.04	0.07	0.12	0.21	0.41	0.80	1.62
states : 240		analysis	0.27	0.54	1.05	2.15	4.28	8.68	17.19	34.80
trans : 720		total	0.30	0.58	1.12	2.27	4.49	9.09	17.99	36.42
	FAM	part.eval.	0.06	0.08	0.09	0.12	0.19	0.31	0.56	1.07
		config.	0.01	0.02	0.05	0.12	0.23	0.42	0.80	1.71
		analysis	0.00	0.01	0.02	0.04	0.08	0.17	0.33	0.70
		total	0.07	0.11	0.16	0.28	0.50	0.90	1.69	3.48
	Perform. Gain		4.29	5.27	7.00	8.11	8.98	10.10	10.64	10.47
Scheduler 7	CMC	total	2.18	4.26	8.62	16.88	34.12	68.49	135.95	276.01
states : 1344	FAM	total	0.49	0.63	0.88	1.34	2.32	4.42	8.39	17.02
trans : 5376	Perfo	orm. Gain	4.45	6.76	9.80	12.60	14.71	15.50	16.20	16.22
Scheduler 9	CMC	total	14.47	28.76	57.34	114.82	233.38	461.56	917.53	1845.39
states : 6912	FAM	total	2.90	3.75	5.20	8.26	14.27	26.61	50.34	99.85
trans : 34560	Perfo	rm. Gain	4.99	7.67	11.03	13.90	16.35	17.35	18.23	18.48

Table 2. Runtime Results.

 $vX(\mu Y([.](A\&X) | Y))$ $vX(\mu Y([\{a,b,c\}](A\&X) | Y))$



Fig. 4. Model and Property for Alternating-Fixpoint Analysis.

 t_i we need k + 1 resettings and recomputations of the inner minimal fixpoint until we reach the solution.

Figure 3 shows the runtime results for the analysis machine FAM and the conventional model checker CMC. The analysis time increases with larger models but the performance gain is constant for a fixed property. This is due to the fact that the initialization phase is almost neglectable here from the very beginning. In fact, the partial evaluation and configuration takes less than 2% of the overall time here.

If we only use 'unparameterized' modalities, which concern all possible transitions, the analysis machine outperforms the conventional model checker by a factor of 11. However, as soon as we use modality parameterization explicitly, the analysis machine outperforms the conventional model checker even by a factor around 30. This is due to the fact that alternating fixpoint analysis requires backtracking with resetting and recomputation of several intermediate results. As the costly selection of the successors of interest is only made once by the analysis machine in the partial evaluation step prior to fixpoint computation, the fixpoint analysis itself nearly takes all the runtime also in this case.

Index of M	any	$\{a,b,c\}$	
M_{500}	CMC	33.84	90.44
states : 503	FAM	2.88	2.90
trans : 505	Performance Gain	11.75	31.19
M_{1000}	CMC	138.51	364.30
states : 1003	FAM	11.64	11.38
trans : 1005	Performance Gain	11.90	32.01
M_{1500}	CMC	312.10	808.08
states : 1503	FAM	26.61	26.52
trans : 1505	Performance Gain	11.73	30.47

Table 3. Runtime Results for Alternating Fixpoints.

Beside these two series of examples we also compared the analysis machine and the conventional model checker on a variety of other applications. The performance gains differ with the complexity of the fixpoint problem. For data flow analysis for example we only achieve factors between 2 and 4 as the problems only require very simple hierarchical or homogeneous fixpoint computations (cf. [KnRS92]). Thus the partial evaluation and specific machine configuration was hardly exploited. But even in these 'worst cases', we still managed to half the computation time. On the other hand, performance gains were very high for computation intensive problems requiring an alternating fixpoint analysis.

7 Conclusions

We have presented a fixpoint-analysis machine, which allows the efficient computation of *homogeneous*, *hierarchical*, and *alternating* fixpoints over *regular*, *context-free/push-down* and *macro* models covering applications that reach from intra- and interprocedural data flow analysis, over model checking for various temporal logics to the verification of behavioural relations between distributed systems. It has turned out that the fixpoint-analysis machine identifies an adequate (parameterized) level for a uniform treatment of all those problems, as it, despite its uniformity, outperforms the 'iteration based' special purpose tools by factors around 10 even if the additional compilation time is taken into account.

We hope that beside its performance, the conceptual structure of the fixpointanalysis machine will also help to improve the understanding of complex fixpoint problems, like e.g. the second order model checking for alternating formulas.

Acknowledgement

The work presented in this paper was strongly influenced by collaboration and intensive discussion with Olaf Burkart, Gerald Lüttgen and Oliver Rüthing. Moreover, the implementation and the embedding in the overall system META-Frame was strongly supported by Carsten Friedrich and Dirk Koschützki.

References

- [Ande92] H. Andersen: "Model Checking and Boolean Graphs", Proc. of ESOP '92, LNCS 582, Springer Verlag, 1992.
- [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 836, pp. 98-113, Springer Verlag.
- [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 V., 1992.
- [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 575, pp.48-58, Springer V.
- [ClSt91b] R. Cleaveland, B. Steffen: "Computing Behavioural Relations, Logically", Proc. ICALP'91, Segovia (Spain), Aug. 1991, LNCS 510, Springer V.
- [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.
- [GKLR94] Geser, A., J. Knoop, G. Lüttgen, O. Rüthing, B. Steffen: "Chaotic Fixed Point Iterations", Tech. Rep. N. MIP-9403, University of Passau (Germany), 1994.
- [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 818, Springer V., pp.169-181.
- [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: "A Tool Kit for Constructing Optimal Interprocedural Data Flow Analyses", Fakultät für Mathematik und Informatik, Univ. Passau, Germany, MIP-Bericht Nr. 9413 (1994).
- [KnSt92a] J. Knoop, B. Steffen: "The Interprocedural Coincidence Theorem", Proc. CC'92, Paderborn (Germany), LNCS N.641, pp. 125-140, Springer V., 1992.
- [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).
- [Lars92] K.G. Larsen: "Efficient Local Correctness Checking", Proc. of CAV'92, Montreal (CAN), LNCS N.663, pp. 410-422, Springer V.
- [MaCS95] T. Margaria, A. Claßen, B. Steffen: "Computer Aided Tool Synthesis in the META-Frame ", 3. GI/ITG Workshop on "Anwendung formaler Methoden beim Entwurf von Hardwaresystemen", Passau (Germany), March 1995, pp. 11-20, Shaker Verlag.
- [Miln89] R. Milner: "Communication and Concurrency", Prentice Hall, 1989.
- [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, BRICS Report N. 94/6, December 1994, pp.466-481.
- [StMC95] B. Steffen, T. Margaria, A. Claßen: "The META-Frame: An Environment for Flexible Tool Management", Proc. TAPSOFT'95, Aarhus, Denmark, May 1995, LNCS N. 915.
- [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.
- [Stef94] B. Steffen: "Finite Model Checking and Beyond", (invited talk) Proc. 6th Nordic Workshop on Programming Theory, Aarhus (Denmark), October 1994, BRICS Report N. 94/6, December 1994, pp. 2-17.
- [StIn94] B. Steffen, A. Ingólfsdóttir: "Characteristic Formulae for Finite State Processes", Information and Computation, Vol. 110, No. 1, 1994.
- [Tars55] A. Tarski: "A Lattice-Theoretical Fixpoint Theorem and its Applications", Pacific Journal of Mathematics, v. 5, 1955.

This article was processed using the LATEX macro package with LLNCS style