The Succinct Solver Suite

Flemming Nielson Hanne Riis Nielson Hongyan Sun
Mikael Buchholtz René Rydhof Hansen Henrik Pilegaard

Informatics and Mathematical Modelling, Richard Petersens Plads bldg. 321, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark
{nielson|riis|sun|mib|rrh|hepi}@imm.dtu.dk

Helmut Seidl

Fakultät für Informatik, I2, TU München, Boltzmannstraße 3, D-85748 Garching, Germany
seidl@in.tum.de

Abstract. The Succinct Solver Suite offers two analysis engines for solving data and control flow problems expressed in clausal form in a large fragment of first order logic. The solvers have proved to be useful for a variety of applications including security properties of Java Card bytecode, access control features of Mobile and Discretionary Ambients, and validation of protocol narrations formalised in a suitable process algebra. Both solvers operate over finite domains although they can cope with regular sets of trees by direct encoding of the tree grammars; they differ in fine details about the demands on the universe and the extent to which universal quantification is allowed. A number of transformation strategies, mainly automatic, have been studied aiming on the one hand to increase the efficiency of the solving process, and on the other hand to increase the ease with which users can develop analyses. The results from benchmarking against state-of-the-art solvers are encouraging.

1 Introduction

Ever since the pioneering work of McAllester [12] there has been a growing interest in using logical formalisms for expressing a variety of control and data flow analyses. This is facilitated by the observation that all polynomial time computable algorithms[1] can be expressed as Horn clauses, and furthermore that the worst case complexity of the specifications can easily be estimated [14]. For problems involving control flow analysis a cubic time bound is inherent although in practice better performance can be obtained in benign cases. In terms of ease of use the logical format separates implementation considerations

[1] Later work, with Ganzinger, deals with logarithmic factors as well.

from specification and hence increases the likelihood that a correct and useful analysis can be developed with only a limited effort. Our work over the last few years has focused on making these insights practical and in testing them on a number of analysis problems that occurred in our other research projects.

To obtain easily readable formulae we quickly decided to go for the "maximal" subset of first order predicate logic that allows the appropriate theoretical results to be established. We thus arrived at Alternation-free Least Fixed Point Logic, ALFP, to be presented in Section 2, that only disallows those features of first order predicate logic that would make it impossible to ensure that a least solution always exists. This is quite in the tradition of Abstract Interpretation where the least solution is guaranteed by a Moore family result [8, 13].

The Succinct Solver Suite encompasses two solver engines (dubbed V1.0 [16] and V2.0 [23]) for computing the least solution as guaranteed by the Moore family result. Additionally there are a number of frontends for clause tuning aiming at increasing performance and ease of use. Other transformations on clauses are by now part of the solvers themselves as well as mechanisms for obtaining feed-back on the internal operation of the solver in order to assist in clause tuning.

A wide variety of applications, to be presented in Section 3, have been used to validate the robustness of the specification language and to suggest the many other features to be provided by the Succinct Solver Suite [16, 23] in order to be a useful tool also for the non-expert. The applications range from familiar programming languages like Java Card, over process calculi like Mobile Ambients, to the study of regular sets of solutions in the context of protocol validation.

Throughout we have focused on increasing the performance by developing a number of syntactic rearrangements of the clauses accepted; many are by now an integral part of the solvers, others are offered through separate front-ends. Equally important has been to relax the rather stringent stratification conditions imposed by the solvers so that users could more easily develop their analyses.

Finally, we have validated the performance of the Succinct Solver Suite against state-of-the-art solvers; the most challenging being XSB Prolog with tabled resolution [21]. We find the results, to be presented in Section 5, encouraging — not least the fact that the Succinct Solver Suite in optimum cases outperforms XSB Prolog by exhibiting a lower asymptotic complexity. On a few cases the Succinct Solver Suite has been able to deal with specifications for which XSB Prolog could not produce a solution.

2 Alternation-free Least Fixed Point Logic

The Alternation-free fragment of Least Fixpoint Logic (ALFP) extends Horn clauses by allowing both existential and universal quantifications in preconditions, negative queries (subject to the notion of stratification), disjunctions of preconditions, and conjunctions of conclusions. The purest approach is to interpret the logic over a universe of unstructured constants but in the interest of flexibility we shall consider ways to allow a finite set of structured ground terms.

2.1 Syntax

Given a fixed countable set $\mathcal{X}$ of variables, a finite set $\mathcal{C}$ of constant symbols, a finite ranked alphabet $\mathcal{R}$ of predicate symbols, and a finite ranked alphabet $\mathcal{F}$ of function symbols - and let us assume that all ranks are at least 1 - we define the set of pre-ALFP clauses, cl, together with preconditions, pre, and terms, t, by the following grammar

$$\begin{array}{lcl} t & ::= & c \mid x \mid f(t_1, \ldots, t_k) \\ pre & ::= & R(t_1, \ldots, t_k) \mid \neg R(t_1, \ldots, t_k) \mid t_1 = t_2 \mid t_1 \neq t_2 \\ & \mid & pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \mid \exists x : pre \mid \forall x : pre \\ cl & ::= & R(t_1, \ldots, t_k) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid \forall x : cl \mid pre \Rightarrow cl \end{array}$$

where $c \in \mathcal{C}$, $x \in \mathcal{X}$, $f \in \mathcal{F}$, and $R \in \mathcal{R}$. Occurrences of $R(\ldots)$ and $\neg R(\ldots)$ in preconditions are also called *queries* and *negative queries*, respectively, whereas the other occurrences are called *assertions* of the predicate R. The pre-defined predicate symbols "$=$" and "$\neq$" are infix operators for *equality* and *inequality* respectively, and we write $\mathbf{1}$ for the always true clause.

In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we introduce a notion of stratification similar to the one which is known from *Datalog* [7, 3]. To express this we make use of a mapping $rank : \mathcal{R} \to \mathbb{N}$ that maps predicate symbols to ranks in $\mathbb{N} = \{0, 1, \ldots\}$. We say that a clause cl is stratified (w.r.t. $rank$) if it has the form $cl = cl_0 \wedge \ldots \wedge cl_k$, and the mapping $rank : \mathcal{R} \to \mathbb{N}$ satisfies the following properties for all $i = 0, \ldots, k$ and $j_i \in \mathbb{N}$:

1. $j_0 < \cdots < j_k$;
2. $rank(R) = j_i$ for every predicate R of assertions in cl_i;
3. $rank(R) \leq j_i$ for every predicate R of queries in cl_i; and
4. $rank(R) < j_i$ for every predicate R of negative queries in cl_i.

(It is natural to choose $j_0 = 0, \ldots, j_k = k$ but the added flexibility makes it easier to make a point later on.) Intuitively, stratification ensures that a negative query is not performed until the relation queried has been fully evaluated.

2.2 Semantics

Let $\mathcal{U}$ denote the universe of ground terms, i.e. terms that do not contain variables. Given interpretations ρ and σ for predicate symbols and terms, respectively, we define the satisfaction relations

$$(\rho, \sigma) \models pre \quad \text{and} \quad (\rho, \sigma) \models cl$$

for preconditions and clauses in the standard way. In particular, we use $\rho(R)$ to stand for the set of k-tuples $(a_1, \ldots, a_k)$ from $\mathcal{U}^k$ associated with the k-ary predicate R and $\sigma(x)$ to stand for the element of $\mathcal{U}$ denoted by the variable x.

We shall mainly be interested in clauses cl that have no free variables. Hence the choice of the interpretation σ is immaterial, so we can fix an interpretation

σ_0 with a finite range. We then call an interpretation ρ of the predicate symbols a solution to the clause cl provided $(\rho, \sigma_0) \models cl$.

Let Δ be the set of interpretations ρ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$, then $\Delta = (\Delta, \sqsubseteq)$ forms a complete lattice, where the lexicographical ordering $\sqsubseteq$ is defined by $\rho_1 \sqsubseteq \rho_2$ if and only if there is some $j \in \mathbb{N}$ such that the following properties hold:

- $\rho_1(R) = \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) < j$
- $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) = j$
- either j is maximal in $rank$ or $\rho_1(R) \subset \rho_2(R)$ for at least one $R \in \mathcal{R}$ with $rank(R) = j$

Proposition 1. *Assume that cl is a stratified pre-ALFP clause without free variables. Then the set $\Delta' = \{\rho \in \Delta \mid (\rho, \sigma_0) \models cl\}$ forms a Moore family, i.e. it is closed under greatest lower bounds.*

In the sequel we shall only be interested in the least solution ρ as guaranteed by the above proposition.

2.3 ALFP in Succinct Solver V1.0 vs. V2.0

The ALFP logic is defined to be the set of pre-ALFP clauses obtained by disallowing function symbols, i.e. by taking $\mathcal{F} = \emptyset$. The least solution ρ for stratified clauses, as well as the universe $\mathcal{U}$, will then be finite and may be computed using the Succinct Solver[2] V1.0 [16].

In the interest of flexibility the Succinct Solver V1.0 admits a slightly larger logic called ALFP-1.0. The rationale is to allow $\mathcal{U}$ to be a finite set of structured ground terms so that one can dispense with the coding tricks that represent k-ary function symbols using $(k+1)$-ary predicates. Syntactically we re-allow function symbols, i.e. $\mathcal{F}$ may be non-empty, but impose the condition that only variables or ground terms may be arguments to assertions in the clauses considered. (Ground terms may still be used as arguments of queries). The universe $\mathcal{U}$ is then defined as the set of all sub-terms of ground terms occurring as arguments to assertions in the clause cl of interest. This ensures that the least solution ρ, as well as the universe $\mathcal{U}$, of a stratified clause remain finite and, hence, may be computed using the Succinct Solver[3] V1.0 [16].

The Succinct Solver V2.0 alleviates the need to precompute the finite universe $\mathcal{U}$, and to represent it using terms in the clause cl considered, at the expense of disallowing universal quantification in preconditions. To be more specific, let the "explored universe" $\mathcal{U}_\star$ be the least subset of $\mathcal{U}$ such that the range of ρ_0 is included in $\mathcal{U}_\star$ and each k-ary predicate R has $\rho(R) \subseteq \mathcal{U}_\star^k$ where ρ is the least solution. Clearly ρ is finite if and only if $\mathcal{U}_\star$ is. The logic ALFP-2.0 is obtained from pre-ALFP, by

[2] The succinct solvers do not syntactically distinguish between variables and constants; instead the constants are taken as the free variables in the clause considered.

[3] Actually it only allows general terms as arguments to queries of the form $x = t$.

- syntactically disallowing universal quantification in preconditions,
- adjusting the semantic interpretation of terms to only operate over $\mathcal{U}_\star$, i.e. a clause like $\forall x : pre \Rightarrow cl[t]$ really means $\forall x : \mathcal{U}_\star(x) \wedge pre \Rightarrow \mathcal{U}_\star(t) \wedge cl[t]$ where $cl[t]$ denotes a clause with a term t occurring as an argument to some assertion.

In the Succinct Solver V2.0 [23] the "explored universe" is expanded dynamically. The solver will terminate and produce the least solution ρ for those stratified clauses cl of ALFP-2.0 for which the least solution ρ (and hence $\mathcal{U}_\star$) as guaranteed by Proposition 1 is indeed finite. (It is possible to adapt termination analyses to safely indicate a set of clauses of ALFP-2.0 for which the least solution is finite but so far these have not been integrated with the Succinct Solver Suite.)

Example 1. The Succinct Solver V2.0 accepts the clause

$$R(a) \wedge \forall x : (R(x) \Rightarrow T(f(x)))$$

and upon termination produces the dynamically expanded universe $\mathcal{U}_* = \{a, f(a)\}$.

In the Succinct Solver V1.0, this clause has to be encoded as the considerably less intuitive

$$R(a) \wedge \forall x : \forall y : (R(x) \wedge (y = f(a)) \Rightarrow T(y))$$

and the universe $\mathcal{U} = \{a, f(a)\}$ must be precomputed before solving starts. □

Example 2. In the analysis of Java Card (to be presented in Section 3.1) we use the predicate S to abstract the run-time stack: $S(m, pc, i, a)$ is supposed to indicate that at a program point pc inside the method m, the stack may contain the element a in position i (using 0 for the top of the stack). Hence the clause

$$\forall i : \forall a : S(m, pc, i, a) \Rightarrow S(m, pc', \mathsf{suc}(i), a)$$

copies stack elements as a preparation for pushing a new element on top of the stack. This formula is directly acceptable for the Succinct Solver V2.0 whereas in V1.0 one needs to write it as

$$\forall y : \forall i : \forall a : y = \mathsf{suc}(i) \wedge S(m, pc, i, a) \Rightarrow S(m, pc', y, a)$$

and to precompute the highest stack position needed; this might take the form of adding a clause with a term

$$\mathsf{suc}(\mathsf{suc}(\ldots(\mathsf{suc}(0))\ldots))$$

corresponding to the maximal height of a stack that can arise during execution. □

2.4 Relationship to Datalog and Prolog

Datalog extends propositional Horn logic with constant and variable symbols and is a commonly used core language for deductive database systems. In the

CORAL system, e.g., this core language is extended with structured terms, non-floundering stratified negation and various extra features such as arithmetic and a native code interface to C++ [20]. In this respect, the ALFP approach is more "puristic". It does not aim at providing a fully fledged programming environment but instead offers a rich and convenient *logical* formalism where the least model is still efficiently computable. It is for this reason that we support explicit scoping of variables, conjunctions in conclusions etc. Accordingly, the base version ALFP-1.0 does also support universal quantification in preconditions. This feature is only meaningful in the presence of a finite universe – where it turns out to be more expressive than Datalog.

Example 3. Consider the clause [16] and an a priori defined edge relation E:

$$\forall x : (\forall y : \neg E(x, y) \vee A(y)) \Rightarrow A(x)$$

Taking $rank(E) = 1$ and $rank(A) = 2$ this ALFP formula defines a predicate A that holds on the set of all acyclic nodes in a graph given by the edge relation E, i.e., all nodes from which no cycle can be reached. Without syntactically expanding the formula to consider each element of $\mathcal{U}$, this predicate is not definable in Datalog (even if extended with stratified negation [10]). □

Prolog extends Datalog with function symbols, negation as failure[4], and various programming constructs and is used in many logic programming systems. However, contrary to the Succinct Solver Suite many Prolog systems may loop infinitely even when only a finite subset, corresponding to $\mathcal{U}_*$, of the Herbrand universe is needed. Also, the depth-first SLD-resolution scheme, according to which Prolog programs are often evaluated, is sometimes inefficient when more than a single solution has to be computed because many subgoals are computed more than once. For these reasons Prolog systems are not in general usable as fixed point engines.

The combination of tabling and Prolog as implemented in XSB Prolog [21] solves these problems because tabling ensures that subgoals are evaluated at most once. The appropriate use of tabling both guarantees termination of programs when $\mathcal{U}_*$ is finite and greatly increases efficiency, thus allowing XSB Prolog to operate as a capable fixed point engine. In Section 5 we present the results of comparative benchmarking of XSB Prolog and the Succinct Solver Suite.

3 Applications

We have used the solver engines of the Succinct Solver Suite on a number of substantial applications as reported below. The general procedure can be outlined as follows:

[4] For some Prolog systems that evaluate according to two-valued semantics the existence of unique least models is only guaranteed for stratified programs.

The first phase of the application is the generation of clauses and this is clearly specific to the application at hand; the second phase is to solve the clauses using the general tool set provided by the Succinct Solver Suite and this may involve some clause tuning to increase performance (see Section 4).

3.1 Safety and Security of Java Card Byte-code

Java Card is a variant of the Java language specifically designed for use in smart cards and other systems with limited resources.

The SecSafe project, cf. [22], has focused on using *static analysis* for verifying safety and security properties for applets written in Carmel, a dialect of the Java Card Virtual Machine Language, JCVML. The analyses developed in SecSafe cover both general features, e.g. control and data flow analyses, as well as features specific to the Java Card platform, e.g. ownership analysis for the on-card *applet firewall*.

Many of the analyses have been implemented by converting the analysis specification into a clause generator for ALFP [9] using the Succinct Solver Suite to solve the clauses. As an example the clauses generated for the control flow analysis of the "`getfield this` f" instruction are shown below, where "$\mathsf{mid}(cls, mth)$" indicates the class and method of the instruction while pc gives the specific program counter of the instruction (and pc' is the program counter of the immediately following instruction):

$$\begin{array}{l}\forall r : \forall a : L(\mathsf{mid}(cls, mth), pc, \mathsf{var\_0}, r) \wedge H(r, f, a) \Rightarrow \\ \quad S(\mathsf{mid}(cls, mth), pc', zero, a) \wedge \\ \forall i : \forall a : S(\mathsf{mid}(cls, mth), pc, i, a) \Rightarrow S(\mathsf{mid}(cls, mth), pc', \mathsf{suc}(i), a) \wedge \\ \forall x : \forall a : L(\mathsf{mid}(cls, mth), pc, x, a) \Rightarrow L(\mathsf{mid}(cls, mth), pc', x, a)\end{array}$$

This specification reflects that the instruction fetches the value of the field f, from the heap H, of the current object (a reference to which is found in local variable 0, $\mathsf{var\_0}$, of the local heap L) and places it on top of the stack S. Operand stacks and local heaps are computed for every instruction and therefore the current stack and local heap, both at program counter pc, are copied forward to the next instruction (located at program counter pc'). The stack contents is also moved down one position to make room at the top. Other parts of the Carmel program give rise to other clauses.

This rather naïve implementation, where no effort is made to optimise the underlying representation, is sufficient even for realistic applets as witnessed by the demonstration applet, called DeMoney, developed for the SecSafe project by Trusted Logic [11]. Solving the clauses generated for DeMoney takes on the order of 30 seconds. The benchmarks for DeMoney are discussed in more detail in Section 5.

3.2 Mobility and Access Control

Modern distributed systems such as wireless internet and mobile telephony have *mobility* of computational entities as a main characteristic. One popular model

of mobility has been put forward in the calculus of *Mobile Ambients* [6]. There, computational entities (both conceptual and physical) are modelled as boundaries called *ambients*. An ambient can be inside another ambient and ambients are, thus, organised in a tree structure where mobility is represented as the capability of an ambient to move *in* and *out* of other ambients.

The ambient tree structure can be represented in ALFP as a binary predicate, I (for *inside*), describing a father-son relationship between ambients. For example, if an ambient named a contains two ambients named b and c, respectively, then it can be stated as the first two facts below:

$$I(a, b) \wedge I(a, c) \wedge I(c, in(b))$$

The third fact states that $in(b)$ is placed inside the ambient c and this represents the capability of the ambient c to move *into* the ambient b.

A control flow analysis (a so-called 0CFA) of Mobile Ambients approximates the ambient movement within the tree structure using the binary predicate I. For example, the execution of the capability to move *into* another ambient may be expressed in ALFP as [18]

$$\forall x : \forall y : \forall z : I(x, y) \wedge I(x, z) \wedge I(y, in(z)) \Rightarrow I(z, y)$$

and reads: if the ambients y and z are both inside the ambient x and, furthermore, y contains the capability to move into z then y may also be inside z as stated in the conclusion. The least solution to this clause in conjunction with the ground facts above is $\rho(I) = \{(a, b), (a, c), (c, in(b)), (b, c)\}$ describing e.g. that c may end up inside b during an execution of the ambient program.

Interestingly, the result of the analysis can also be used to ensure absence of movements. For example, the above result shows that the ambient a cannot access the ambient b. This relates to the area of *access control* as studied for example in [18] where the calculus of Discretionary Ambients is presented along with two control flow analyses. One analysis is a 0CFA as for Mobile Ambients that approximates a father-son relationship while the other is a 1CFA that approximates a grandfather-father-son relationship represented by a ternary relation. The analyses of Discretionary Ambients have been used to study mandatory access control as well as discretionary access control and have also served as the basis for extensive experiments with the Succinct Solver Suite as reported in Sections 4 and 5.

3.3 Cryptographic Protocols

There is a long and successful tradition of analysing cryptographic protocols that relies on modelling (perfect) cryptography as structured *terms*. For example, a message m encrypted under a key k can be modelled as the term $e(k, m)$, where e is a binary constructor, and decryption can be modelled as a corresponding destructor. This is e.g. the approach taken in the Spi-calculus [2] and LySa [4] and their control flow analyses.

Consider a process, P, that repeatedly receives a message on the network in a variable x and sends $e(k, x)$ back onto the network. If it is placed in a context such that it initially receives the message m, then x will be bound to all the elements from the *infinite* set $S = \{m, e(k, m), e(k, e(k, m)), e(k, e(k, e(k, m))), \cdots\}$ during the execution of P. Hence the least solution cannot be finite.

In [4] this problem is solved by representing the infinite sets of terms using their generating tree grammars. The rules of the grammar are represented by a binary predicate R. An encoding of the grammar rules of the above set S, for example, gives rise to the facts:

$$R(l_1, e(l_2, l_1)) \quad \wedge \quad R(l_1, m) \quad \wedge \quad R(l_2, k)$$

where the first argument denotes the left-hand side of a grammar rule and the second argument denotes corresponding right-hand side. Set operations, such as membership, subset, etc., of these infinite set of terms can then be encoded in ALFP as manipulations of the grammar rules in R.

Overall, the analyses can be implemented in polynomial-time in the size of the program and in [4] the analysis is shown to be sufficiently precise to identify well-known attacks on a number of symmetric key protocols as well as showing the correctness of their amendments.

4 Program Transformations

The approach taken in the development of the succinct solvers has been to obtain a generic tool aiming at achieving the best asymptotic worst case performance reported for any analysis engine [16]. Therefore the only way to influence the operation of the solver is to perform clause tuning, which amounts to transforming the clause given as input. We have aimed at ensuring that the beneficial rearrangements can be understood at the level of inspecting the clauses themselves; this is contrary to, and in our opinion more user friendly, than the approach taken in some other systems where the user is supposed to make intelligent choices about the internal operation of the solver concerning e.g. iteration strategies.

Typically clause tuning is performed by using the clause to solve only small problems and, based on the sizes of the predicates computed, to rearrange the clause so as to be more efficient also for large problems. One additional feature has proved very helpful: one can instruct the solver to report the number of environments η (values for variables) propagated across selected implications. In Succinct Solver V2.0 [23] this is written as $pre \Longrightarrow cl$ and we have found that information about the number of environments gives useful information about where the solver spends its time.

The Order of Conjuncts in Preconditions. In the succinct solvers preconditions in implications are evaluated from left to right and in the context of an environment η that describes successful bindings of variables. When checking a query to a predicate P the evaluation of the remainder of the precondition is

performed for all new environments η' that are obtained by unifying η with an element currently in P. The unification will fail when the binding of the variables in η does not coincide with the element of P and in this case, no further work is done. Thus, we may expect to gain efficiency by making the unification fail as early as possible in the evaluation of a precondition.

Example 4. Consider the clause $\forall x : P(x, a) \wedge P(b, x) \Rightarrow Q(x)$ Suppose that we have *a priori* knowledge that P contains few elements with b as the first component but many elements with a as the second component. Then, swapping the two conjuncts will increase efficiency since the clause

$$\forall x : P(b, x) \wedge P(x, a) \Rightarrow Q(x)$$

will have fewer environments propagated from the first query to P. □

This observation leads to the manual optimisation strategy that queries, which restrict the variable binding most, should be put at the beginning of preconditions. Experiments with our analysis of Discretionary Ambients [5] have shown that reordering of conjuncts in preconditions may significantly improve the efficiency of solving otherwise identical clauses. As expected, the increase in efficiency varies with the structure of programs and typically ranges from a factor of ten up to a decrease in the degree of the complexity polynomial.

Memoisation. We apply *memoisation* techniques to avoid propagation of identical environments in the succinct solvers. The propagation of *completely* identical environments can only occur at disjunction and at existential quantification and hence the solver includes an automatic memoisation scheme for these constructs only.

Example 5. Consider the clause

$$\forall x : \forall y_1 : \forall y_2 : \forall y_3 : P(x, y_1) \wedge Q(y_1, y_2, y_3) \Rightarrow R(y_1, y_2, y_3)$$

where all the environments established when querying P will contain x and are propagated although x is used neither in the query to Q nor in the conclusion. If two of these environments are identical, except for the value of x, then the remainder of the clause will be evaluated twice with the exact same result.

In this case we may expect to gain efficiency by manually transforming clauses using existential quantifications or disjunctions:

$$\forall y_1 : \forall y_2 : \forall y_3 : (\exists x : P(x, y_1)) \wedge Q(y_1, y_2, y_3) \Rightarrow R(y_1, y_2, y_3)$$

Here the universal quantification of x is transformed into an existential quantification in the precondition. Thus the *memoisation* scheme ensures that no identical environments are propagated. □

Weak Stratification. We now survey recent work [15] aiming at relaxing the rather stringent demands imposed by stratification. Somewhat informally weak stratification relaxes the restrictions imposed by stratification conditions (1) and (2) described in Section 2.1: the condition (1) is not imposed any longer and condition (2) is replaced by the requirement that $rank(R) \geq j_i$ for every predicate R of assertions in cl_i. In practice, this makes it much more convenient to write easily readable specifications.

Example 6. The clause $\forall x : R(x) \Rightarrow (S(x) \wedge T(x))$, where $rank(R) = 1$, $rank(S) = 2$ and $rank(T) = 3$, is weakly stratified but not stratified. □

This transformation is available as a preprocessor in the Succinct Solver Suite.

5 Benchmarks

The use of logic based formalisms for the specification of control flow and data flow problems enables a convenient separation of concerns between specification and implementation [17]. This allows the implementation of a given analysis to be based on general purpose fixed point engines such as those of the Succinct Solver Suite.

While the worst-case complexity of, e.g., control flow analyses is inherently cubic and largely independent of the actual fixed point engine, better performance may be obtained in practice for a large family of benign programs. Hence the original solver was designed to give state-of-the-art asymptotic worst-case performance while allowing for even better performance in benign cases [16].

In [19] we compare the performance of the Succinct Solver Suite (solver V2.0) to that of XSB Prolog V2.6 [21]. The reported results are based on benchmarks obtained from control flow analyses both of Discretionary Ambient programs (Section 3.2) and of Carmel programs (Section 3.1).

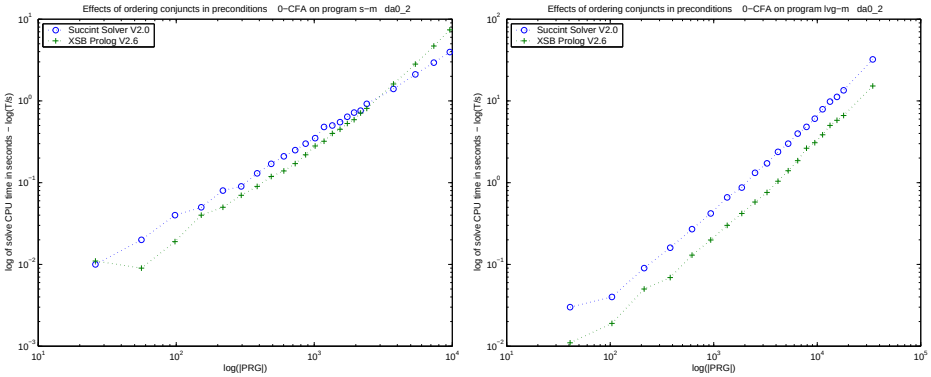
When running the experiments we timed the initialisation and solve phases of the solvers separately. To do this we executed the Succinct Solver Suite in two stages (initialisation, solve) and used pre-compiled programs for XSB Prolog (compilation, solve). Where possible times were collected with and without garbage collection and the algorithm comparison is based on the times without garbage collection. We used ALFP clause generators but fed identical Normal programs[5], obtained by a syntactical expansion of ALFP clauses into logically equivalent Normal clauses, to the two solvers.

The Discretionary Ambient programs constitute abstract descriptions of a matrix-like grid of routers in which a packet has to travel from one end to the other. This structure is convenient as the complexity of the corresponding analysis problems can be adjusted by changing the connectivity of the underlying graph, i.e. increasing or decreasing the number of sites reachable in one step.

We find that problems can be divided into two types. The first type of problems induce lightly populated analyses, i.e. the order of the ratio between the

[5] Normal programs extend Definite (Horn) programs by allowing both positive and negative literals in clause bodies, i.e. allowing both positive and negative queries.

Running time without garbage collection and initialisation as function of program size. Logarithmic Scales.



Lightly populated analysis. The Succinct Solver Suite shows lower asymptotic complexity.

Heavily populated analysis. XSB Prolog shows constant factor advantage.

Fig. 1. Benchmarks of scalable Discretionary Ambient programs.

size of the computed interpretation, $|\rho|$, and the size of the finite universe over which it is computed, $|\mathcal{U}_*|$, is $\mathcal{O}(1)$. For these problems the Succinct Solver Suite outperforms XSB Prolog by having a substantially lower asymptotic complexity on optimum clauses as depicted on the left in Figure 1.

The other type of problems induce heavily populated analyses, i.e. the order of the aforementioned ratio is more like $\mathcal{O}(\sqrt[k]{|\mathcal{U}_*|})$ for some $k \neq 0$. For these problems the Succinct Solver Suite at worst performs a small constant factor worse than XSB Prolog as depicted on the right in Figure 1.

The Carmel programs are derived from the DeMoney case study [11] provided by the industrial partner, Trusted Logic, of the EU project SecSafe. This demonstrative electronic Java Card purse was provided along with a partial Carmel implementation of V2.12 of the Java Card API. In terms of size these two programs are both small to medium programs while their combination (Applet+API) is a fairly large smart card program (about 6700 lines of code).

As seen in Table 1 the Succinct Solver Suite is slightly slower for the smallest program but twice as fast for the largest - in terms of CPU-time used for the fixed point computation. In terms of total time used the solvers perform within 10% of one another - with a marginal advantage to the Succinct Solver Suite. Interestingly, the two systems spend their time differently. XSB Prolog spends much time in the initialisation phase while the Succinct Solver Suite spends much time garbage collecting. Given that memory was never exhausted, this behaviour from the garbage collector is a bit surprising and we attribute it to the garbage collection policy of New Jersey SML, in which the Succinct Solver Suite is implemented.

Solver	Input/Mb	Sol. Size	solve CPU/s	Init/s	Total/s
Demoney Applet Stand-Alone					
Succinct Solver V2.0	1.2	4409	2.11	4.59	8.77
XSB Prolog V2.6	1.2	4409	1.78	8.27	10.41
Partial jc212 API (Trusted Logic)					
Succinct Solver V2.0	1.2	7734	2.71	4.81	11.37
XSB Prolog V2.6	1.2	7734	2.75	8.31	11.44
Combination of Applet and API					
Succinct Solver V2.0	2.4	15479	6.36	10.16	28.86
XSB Prolog V2.6	2.4	15479	13.37	16.25	30.24

Table 1. Benchmarks of Carmel based programs (2GHz Pentium 4).

6 Conclusion

The Succinct Solver Suite offers two analysis engines for solving data and control flow problems expressed in clausal form. Version 1.0 admits general clauses in a slight superset of Alternation-free Least Fixed Point Logic (ALFP) subject to a notion of stratification; intuitively ALFP is the largest fragment of first order logic that admits proving the existence of solutions over finite unstructured universes. Version 2.0 further restricts the clauses by disallowing universal quantification in preconditions; this facilitates extending the solver technology to operate over "finitely explored" structured universes.

Many applications are equally suited for versions 1.0 and 2.0 but each have applications where it is more suited than the other. In the case of version 1.0 the use of universal quantification in preconditions is indispensable when developing static analyses with mixed modalities. In the case of version 2.0 a typical example is the Java world where stack sizes are going to be finite for well-formed programs and hence one should like to avoid the a priori calculation of an upper bound.

The solvers have proved to be useful for a variety of applications. This includes analysing security properties of programs in Java Card byte-code. Variations of Mobile Ambients, in particular Safe and Discretionary Ambients, have proved useful for formulating discretionary and mandatory access control policies in the world of mobility, and the solvers have proved quite effective in solving the relevant analysis questions. Finally, we have applied the solvers to the problem of validating confidentiality and authenticity properties of protocol narrations formalised in a suitable process algebra. Ongoing work explores the use of the Succinct Solver Suite for analysing the hardware programming language VHDL as well as biologically inspired process algebras.

It is evident that the demands placed by the solver on clauses in order to achieve good efficiency, and the wishes of the user in order to develop easily readable clauses in a systematic manner, may be contradictory. To this effect we have studied and implemented a number of transformation strategies that are aimed at increasing the efficiency of the solver or at presenting a more flexible interface to the user.

The Succinct Solver Suite has been benchmarked against other solvers, mainly XSB Prolog with tabled resolution. The performance of the Succinct Solver Suite

is at worst a small constant factor worse than XSB Prolog, which is hardly surprising given that the Succinct Solver Suite is written in Standard ML and spends a lot of time on garbage collection, whereas XSB Prolog is a heavily optimised C program. What is more interesting is that in optimum cases the Succinct Solver Suite outperforms XSB Prolog by having a substantially lower asymptotic complexity. On the SecSafe benchmark program, DeMoney, the two solvers exhibit the same running times (within a 10% margin). On a few cases the Succinct Solver Suite has been able to deal with specifications for which XSB Prolog could not produce a solution [1].

In future developments we hope to further assist clause tuning by incorporating the techniques for automatic estimation of the sizes of predicates [14] — something which is only feasible due to the very predictable behaviour of the solver engines. Further, we hope to provide a feature that allows users to enforce a full stop of the solver engine at the point of failure of a so-called observation predicate. The finite counter-example usually implied by such a failure is easier to recover from the partial result present at this point of computation. Finally, we hope to include the possibility of creating fresh and unique names at will during the course of computation. This would facilitate applications where the purpose of the analysis is to construct a finite automaton characterising the solution.

The Succinct Solver Suite is available from our web-pages and may be freely used for research and development.

http://www.imm.dtu.dk/cs_SuccinctSolver

Acknowledgements This work has been supported by the EU-projects SecSafe (IST-1999-29075) and DEGAS (IST-2001-32072) as well as the Danish Natural Science Research Council project LoST (21-02-0507).

References

1. Personal communication with Luis Fernando P. de Castro from the XSB team. `http://sourceforge.net/mailarchive/message.php?msg_id=4349555`.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols – The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.
3. K. Apt, H. Blair, and A. Walker. A theory of declarative programming. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufman, 1988.
4. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140. IEEE Computer Society Press, 2003.
5. M. Buchholtz, F. Nielson, and H. Riis Nielson. Experiments with Succinct Solvers. Technical Report IMM-TR-2002-4, Informatics and Mathematical Modelling, Technical University of Denmark, 2002.
6. L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

7. A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'97*, pages 238–252. ACM Press, New York, NY, 1977.
9. R. Rydhof Hansen. A prototype tool for JavaCard firewall analysis. In *Nordic Workshop on Secure IT-Systems, NordSec'02*, Karlstad, Sweden, November 2002. Proceedings published as Karlstad University Studies 2002:31.
10. P. G. Kolaitis. Implicit definability on finite structures and unambiguous computations (preliminary report). In *5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 168–180, 1990.
11. R. Marlet. DeMoney: Java Card implementation. SECSAFE-TL-008, Trusted Logic, November 2002.
12. D. McAllester. On the complexity analysis of static analyses. In *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 312–329, 1999.
13. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
14. F. Nielson, H. Riis Nielson, and H. Seidl. Automatic Complexity Analysis. In *European Symposium on Programming (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer Verlag, 2002.
15. F. Nielson, H. Riis Nielson, and H. Sun. Observation predicates in Flow Logic. Secsafe-imm-010, Informatics and Mathematical Modelling, Technical University of Denmark, September 2003.
16. F. Nielson, H. Seidl, and H. Riis Nielson. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9:335–372, 2002.
17. H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Verlag, 2002.
18. H. Riis Nielson, F. Nielson, and M. Buchholtz. Security for mobility. In *Proceedings of FOSAD 2001*, volume 2946 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
19. H. Pilegaard. A feasibility study - the Succinct Solver v.2.0, XSB Prolog v.2.6, and flow-logic based program analysis for Carmel. SECSAFE-IMM-008, Informatics and Mathematical Modelling, Technical University of Denmark, October 2003.
20. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. *VLDB Journal*, 3(2):161–210, 1994.
21. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB System. Web page: `http://xsb.sourceforge.net/`, 2003.
22. I. Siveroni. SecSafe. Web page: `http://www.doc.ic.ac.uk/~siveroni/secsafe/`, 2003.
23. H. Sun, H. Riis Nielson, and F. Nielson. Extended features in the Succinct Solver (V2.0). SECSAFE-IMM-009, Informatics and Mathematical Modelling, Technical University of Denmark, October 2003.