Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 57–66
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

DATALOG_SOLVE: A Datalog-Based Demand-Driven Program Analyzer [1]

M. Alpuente M. A. Feliú C. Joubert A. Villanueva [2]

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain

Abstract

This work presents a practical Java program analysis framework that is obtained by combining a Java virtual machine with a general-purpose verification toolbox that we previously extended. In our methodology, Datalog clauses are used to specify complex interprocedural program analyses involving dynamically created objects. After extracting an initial set of Datalog constraints about the Java bytecode program semantics, our framework transforms the Datalog rules of a particular analysis into a Boolean Equation System (BES), whose local resolution using the aforementioned extended verification toolbox corresponds to the demand-driven computation of the analysis.

Keywords: Java program analysis, Datalog, boolean equation system, demand-driven evaluation

1 Introduction

Static program analysis extracts semantic information from a given program without running it. An example of such an analysis is the definition-use analysis that is used to analyze data-flow program dependencies. The analysis is run on an abstract representation of the program that contains the variable definitions as well as their use at each program statement.

In this work, we focus on static *reference* analyses of Java programs and, more generally speaking, on any object-oriented programming language that is characterized by data abstraction, inheritance, polymorphism, dynamic binding of method calls and/or dynamic loading of classes. A reference analysis, also called *points-to* analysis, determines information about the set of objects to which a reference variable or field may point during the program execution. There is a real interest in

[1] This work has been supported by the Spanish MEC/MICINN under grant TIN 2007-68093-C02-02, by the Generalitat Valenciana GVPRE/2008/113, and by the Universidad Politécnica de Valencia, under grant PAID-06-07 (TACPAS).

[2] Email: {alpuente,mfeliu,joubert,villanue}@dsic.upv.es

1571-0661/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2009.07.059

using such kind of analysis in program understanding tools (e.g. semantics browsers or program slicers), in software maintenance tools and also in testing tools that rely on coverage metrics.

Recently, a large number of rule-based specifications of program analyses have been developed in a simple relational query language called Datalog, see [10,13]. This language has shown to be rich enough to describe complex interprocedural program analyses involving dynamically created objects.

The advantages of formulating dataflow analyses as a Datalog query are twofold. On the one hand, analyses that take hundreds of lines of code in a traditional language can be expressed in a few lines of Datalog [13]. On the other hand, a number of important optimization techniques for Datalog have been studied in the areas of logic programming and deductive databases [1,4]. Unfortunately, even after those optimizations, direct implementations of the analyses based on logic programming are often slower than the corresponding imperative versions. Recently, a very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) has been developed in the BDDBDDB system [13], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach.

In this work, we present DATALOG_SOLVE, which is a fully automatic and efficient demand-driven Datalog-based program analyzer developed within the CADP verification toolbox [6]. DATALOG_SOLVE analyses the satisfiability of a Datalog query that represents the points-to analysis on a considered Java program. The front-end of DATALOG_SOLVE encodes a set of Datalog rules (a particular analysis) in terms of a Boolean Equation System (BES) [3] implicitly described by its successor function. The analysis is run on a set of Datalog constraints (facts) which are automatically extracted from Java source code by using the JOEQ compiler framework [12]. The back-end of our tool carries out the demand-driven generation, resolution and interpretation of the BES by means of the generic CÆSAR_SOLVE [8] library of CADP, devised for on-the-fly BES resolution and diagnostic generation. This architecture clearly separates the implementation of Datalog-based static analyses from the resolution engine, which can be extended and optimized independently.

1.1 Related work

The description of data-flow analyses as a database query was pioneered by Ullman [10] and Reps [9], who applied Datalog's bottom-up magic-set implementation to automatically derive a *local* implementation.

Recently, a very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) has been developed in the BDDBDDB system [13], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach. The computation is achieved by a fixed point computation starting from the everywhere false predicate (or some initial approximation based on Datalog facts). Datalog rules are then applied in a bottom-up manner until saturation is reached, so that all solutions satisfying each relation of a Datalog program are exhaustively computed. These sets of solutions are then used to answer complex formulas.

In contrast, to solve a set of queries with no *a priori* computation of the deriv-

able atoms, our approach focuses on demand-driven techniques. In the context of program analysis, note that all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results. Therefore, improvements to top-down evaluation remain attractive for program analysis applications. Recently, Zheng and Rugina [14] showed that demand-driven CFL-reachability with worklist algorithm can compare favorably with an exhaustive solution, especially in terms of memory consumption. Our technique to solve Datalog programs based on local BES resolution goes towards the same direction and provides a novel approach to demand-driven program analyses.

Plan of the work.

The remainder of this article is organized as follows. In Section 2, we introduce the Datalog language and its adequacy to specify points-to analyses. Section 3 describes the translation of a demand-driven evaluation of Datalog queries in terms of boolean equation systems, as well as the tool architecture. A set of experimental results obtained with DATALOG_SOLVE on large Java programs from Sourceforge are given in Section 4. Finally, we draw some conclusions and describe several lines of future work in Section 5.

2 Datalog specification of a program analysis

The Datalog approach to static program analysis [13] can be summarized as follows. Each program element, namely variables, types, code locations, and function names, are grouped in their respective *domains*. By considering only finite program domains, Datalog programs are ensured to be *safe* (i.e., query evaluation only generates a finite set of answers). Each program statement is decomposed into *basic program operations*, namely load, store, assignment, and variable declarations. Each kind of basic operation is described by a relation within the Datalog program. A program operation is then described as a set of tuples satisfying the corresponding relation. In this framework, a *program analysis* consists in either querying extracted relations or computing new relations from existing ones. Let us show an example of analysis specification in our approach.

Example 2.1 Consider the Datalog program (`pa.datalog` [13]) that defines a specific context-insensitive points-to analysis given in Figure 1. The program consists of three parts:

(i) A declaration of *domains* where domain names and sizes (number of elements) are specified.

(ii) A list of *relations*, *i.e.*, predicates, specified by a predicate symbol, its arguments over specific domains and whether it is derived from an applicable Datalog rule (value `outputtuples`), or extracted from the program bytecode (value `inputtuples`).

(iii) A finite set of Datalog *rules*, defining the `outputtuples` relations.

```
### Domains

V   262144   variable.map
H   65536    heap.map
F   16384    field.map

### Relations

vP_0     (variable : V, heap : H)            inputtuples
store    (base : V, field : F, source : V)   inputtuples
load     (base : V, field : F, dest : V)     inputtuples
assign   (dest : V, source : V)              inputtuples
vP       (variable : V, heap : H)            outputtuples
hP       (base : H, field : F, target : H)   outputtuples

### Rules

vP (v, h)         :- vP_0 (v, h).
vP (v1, h)        :- assign (v1, v2), vP (v2, h).
vP (v2, h2)       :- load (v1, f, v2), vP (v1, h1), hP (h1, f, h2).
hP (h1, f, h2)    :- store (v1, f, v2), vP (v1, h1), vP (v2, h2).
```

Fig. 1. Datalog specification of a context-insensitive points-to analysis

In the example of program analysis given in Figure 1, domain `V` represents local variables and method parameters while domain `H` represents heap objects. Finally, domain `F` is used for field identifiers. These domains are extracted from the Java bytecode.

In the second part of the program, the relations `vP_0`, `store`, `load` and `assign` are labeled as `inputtuples` since they are extracted from the Java bytecode (by using a modified version of the JOEQ compiler). For example, the relation `vP_0` consists of initial points-to relations `(v,h)` of a program, *i.e.*, `vP_0(v,h)` is extracted as a fact of the Datalog program if there exists a direct assignment within the Java program between a reference to a heap object `h` $\in$ `H` and a variable `v` $\in$ `V` (*e.g.*, along the Java program occurs a statement `v = new String()`). The other Datalog constraints (`inputtuples`) are computed similarly. Relations `vP` and `hP` are labeled as `outputtuples` since they are inferred from the rules.

The last part of the program specifies which kind of information are we able to infer by using the information extracted from the program. In this case, the four rules infer possible points-to relations from local variables and method parameters (Datalog variables of the domain `V`) to heap objects (Datalog variables of the domain `H`), as well as possible points-to relations between heap objects through field identifiers (Datalog variables of the domain `F`). In summary, the rules model the effect of the input relations over the heap.

Finally, a Datalog query consists of a set of goals over the relations defined in

the Datalog program, *e.g.*, `:- vP(x,y).`, where `x` and `y` are variable arguments of `vP`. This goal aims at computing the complete set of variables `x` that may point to any heap object `y` at any point during the program execution. The Datalog query is not specified in the Datalog program, but provided independently by the user.

3 BES evaluation of a Datalog query

In this section we describe DATALOG_SOLVE, our Datalog query evaluation framework. As we can see in Figure 2, the resolution engine *Datalog solver* takes three inputs: (i) a set of Datalog facts (the *input relations*), which represent the information relevant for the analysis and are automatically extracted from the Java program by means of a compiler (in the case of Java programs we use a modified version of the JOEQ compiler [12]) [3], (ii) the analysis specification consisting of the set of Datalog rules defining the *output relations*, and (iii) the analysis invocation consisting of a set of Datalog goals (Datalog rules with empty head). As we have already said, the domain definition states the possible values for each predicate's argument in the query.

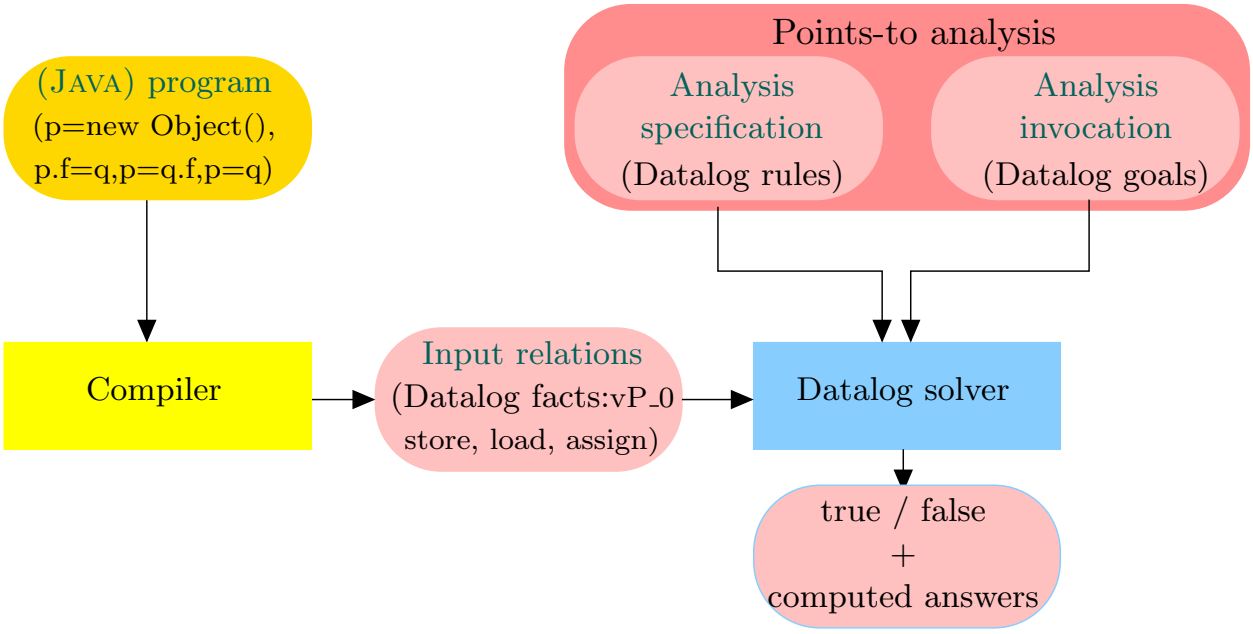

Fig. 2. DATALOG_SOLVE framework for Java program analysis.

The *Datalog solver* in the figure is the resolution engine that checks the satisfiability of the Datalog query and/or extracts the solutions to that query. Thus, the output of the tool when the query is not ground is the set of instances of the query that are satisfied within the Datalog program. Let us now show more in detail how the *Datalog solver* has been implemented.

DATALOG_SOLVE (120 lines of LEX, 380 lines of BISON and 3 500 lines of C code) proceeds in two steps: 1) translation of the Datalog query to BES (*Datalog query representation*), and 2) generation and interpretation of the solutions to the query (*solutions extraction*). We can see in Figure 3 the process.

[3] The domain definitions are also extracted by JOEQ and provided to the resolution engine.

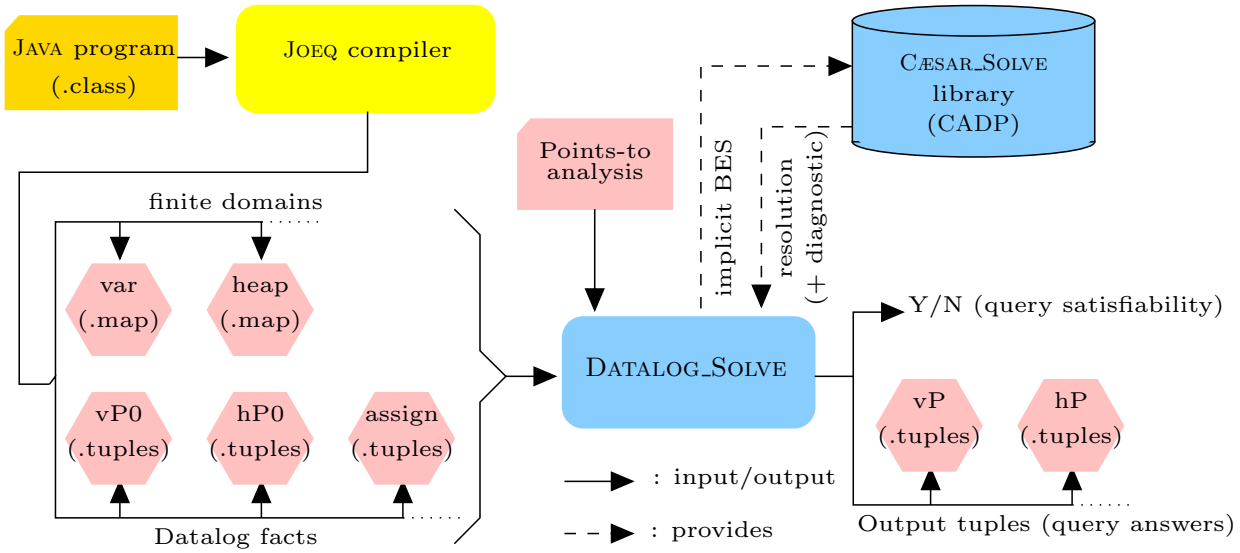

Fig. 3. DATALOG_SOLVE implementation.

First of all, the JOEQ compiler generates two kinds of files: `.map` files representing the domains of Datalog variables, and `.tuples` files, one for each relation, representing the facts of the Datalog program (the information regarding the analysis extracted from the Java program).

The DATALOG_SOLVE engine takes these files, together with the points-to analysis, and generates an implicit BES that represents the Datalog program. The implicit BES is provided to the CÆSAR_SOLVE library that is available within the CADP toolbox and is able to solve it. Finally, our DATALOG_SOLVE interprets and explores the solutions provided by the CÆSAR_SOLVE library, and then generates the output files. In the case of executing a ground query, the output is simply yes or not; otherwise DATALOG_SOLVE generates a set of files `.tuples`, one for each non-ground goal, representing all the instantiations of these goals satisfied by the program.

In the following subsections, we illustrate how the Datalog program representing the points-to analysis can be transformed into an implicit BES. For a more detailed explanation, the reader can consult [2].

3.1 Datalog query representation

DATALOG_SOLVE uses an elegant and direct intermediate representation of the Datalog query (rules and goals) as an implicit BES parameterised with typed boolean variables. Let us illustrate this transformation by means of an example.

Example 3.1 Consider again the Datalog program given in Figure 1 that defines a context-insensitive points-to analysis (`pa.datalog` [13]). The BES transformation of the Datalog-based program analysis for the goals `vP(x,y)` and `hP(z1,w,z2)` consists in the following equation system:

$$x_0 \overset{\mu}{=} \texttt{vP(x,y)} \vee \texttt{hP(z1,w,z2)}$$

$$\texttt{vP(v:V,h:H)} \overset{\mu}{=} \texttt{vP_0(v,h)} \vee$$

$$\begin{array}{rcl} & & \texttt{(assign(v, v2)} \wedge \texttt{vP(v2,h))} \ \vee \\ & & \texttt{(load(v1,f,v)} \wedge \texttt{vP(v1,h1)} \wedge \texttt{hP(h1,f,h))} \\ \texttt{hP(h1:H,f:F,h2:H)} & \stackrel{\mu}{=} & \texttt{store(v1,f,v2)} \wedge \texttt{vP(v1,h1)} \wedge \texttt{vP(v2,h2)} \end{array}$$

Boolean variable x_0 encodes the set of Datalog goals, whereas (parameterised) boolean variables `vP(v:V,h:H)` and `hP(h1:H,f:F,h2:H)` represent the set of Datalog rules in the program. Note that, since the predicate `vP` is defined by three rules, the corresponding boolean variable `vP(v:V,h:H)` is defined as three disjunctions of conjunctions, one for each Datalog rule.

Formally, being G the set of Datalog goals and R the set of Datalog rules, the transformation is defined as follows:

$$(1) \qquad x_0 \stackrel{\mu}{=} \bigvee_{:-\ p_1(d_1),\ \ldots,\ p_m(d_m) \in G} \bigwedge_{i:=1}^{m} p_i(d_i)$$

$$(2) \quad p(d:D) \stackrel{\mu}{=} \bigvee_{p(d)\ :-\ p_1(d_1),\ \ldots,\ p_m(d_m) \in R} \bigwedge_{i:=1}^{m} p_i(d_i)$$

Boolean variable x_0 encodes the set of Datalog goals G, whereas (paremeterised) boolean variables $p(d:D)$ represent the set of Datalog rules R. Since the Cæsar_Solve library requires as input a parameterless Bes, we can obtain it by applying the instantiation algorithm of Mateescu [7] on the implicit Pbes representation. However, the direct transformation resulted too expensive. Therefore, the final parameterless Bes used in our implementation is an optimized version inspired by the Query-Sub-Query optimization for Datalog of [11]. The idea is that variables are instantiated to values only when they are shared by more than one subgoal in the body of a rule. This optimization generates less boolean variables than with the basic, direct approach. The next example clarifies the intuition behind the final transformation (see [2] for details).

Example 3.2 Let us consider the implicit representation in the Example 3.1. Assume the following carrier for the variable domains: carrier(`H`) = {`heap1`,`heap2`}, carrier(`F`) = {`a`,`b`}, and carrier(`V`) = {`var1`,`var2`}. Then, the parameterless representation for the boolean variable `hP(h1:H,f:F,h2:H)` is the following:

$$x^r_{\texttt{hP(h1:H,f:F,h2:H)}} \stackrel{\mu}{=} r_{\texttt{store(v1,f,v2), vP(v1,h1), vP(v2,h2)}}$$

$$\begin{array}{rcl} r_{\texttt{store(v1,f,v2), vP(v1,h1), vP(v2,h2)}} & \stackrel{\mu}{=} & r^{pc}_{\texttt{store(var1,f,var1), vP(var1,h1), vP(var1,h2)}} \vee \\ & & r^{pc}_{\texttt{store(var1,f,var2), vP(var1,h1), vP(var2,h2)}} \vee \\ & & r^{pc}_{\texttt{store(var2,f,var1), vP(var2,h1), vP(var1,h2)}} \vee \\ & & r^{pc}_{\texttt{store(var2,f,var2), vP(var2,h1), vP(var2,h2)}} \end{array}$$

$$\begin{array}{rcl} r^{pc}_{\texttt{store(var1,f,var1), vP(var1,h1), vP(var1,h2)}} & \stackrel{\mu}{=} & x_{\texttt{store}_{\texttt{var1,f,var1}}} \wedge \\ & & x_{\texttt{vP}_{\texttt{var1,h1}}} \wedge \\ & & x_{\texttt{vP}_{\texttt{var1,h2}}} \end{array}$$

$$r^{pc}_{\texttt{store(var1,f,var2), vP(var1,h1), vP(var2,h2)}} \stackrel{\mu}{=} \ldots$$

Boolean variables r instantiate variables in the body of the Datalog rules to its possible values, returning a disjunction of all the possible combinations of variable instantiations. However, it does not instantiate all the variables (namely variables `f`, `h1` and `h2`). Note that all the variables in the body of the rule would have been instantiated with the direct transformation. Boolean variables r^{pc} check whether all the instantiated subgoals in the query of the rule are satisfied. Note that the Cæsar_Solve library generates these equations on-the-fly by using very efficient algorithms for Bes, thus it does not generate unnecessary boolean variables.

3.2 Solutions extraction

The back-end of our system carries out the demand-driven generation, resolution and interpretation of the Bes by means of the generic Cæsar_Solve library of Cadp [6], devised for local Bes resolution and diagnosis.

The tool takes as a default query the computation of the least set of facts that contains all the facts that can be inferred by using the rules that define the program analysis. This represents the worst case of a demand-driven evaluation, where all the information derivable from a Datalog program is computed.

Considering the parameterless Bes illustrated above, the query satisfiability problem is reduced to the on-the-fly resolution of boolean variable x_0. The value computed for x_0 indicates whether there exists at least one satisfiable goal in G. In order to deduce all the different solutions of a given Datalog query, it is necessary to use a breadth-first search (Bfs) strategy for x_0 to force the solver to compute all the solutions for the query. Since the Cæsar_Solve library offers an optimized depth-first search (Dfs) strategy for the kind of system generated, we chose this strategy to solve all $p(d : D)$ boolean variables. Upon termination of the Bes resolution, query solutions are all the boolean variables whose value is true. Datalog_Solve subsequently converts them into *tuples* (combinations of variable values, one for each atom of the query, that lead to a satisfied query) represented numerically in an output file. Datalog_Solve also allows for an iterative enumeration of the solutions with symbolic value names instead of numbers mainly for debugging purposes.

4 Experimental Results

The Datalog_Solve framework was applied to a number of Java programs by computing the context-insensitive pointer analysis described in Figure 1.

Table 1
Description of the Java projects used as benchmarks.

Name	Description	Classes	Methods	Bytecodes	Vars	Allocs
freets	speech synthesis system	215	723	46K	8K	3K
nfcchat	scalable, distributed chat client	283	993	61K	11K	3K
jetty	server and servlet container	309	1160	66K	12K	3K
joone	Java neural net framework	375	1531	92K	17K	4K

To evaluate the scalability and applicability of the transformation, we applied our technique to four of the most popular 100% Java projects on Sourceforge that could compile directly as standalone applications. These projects were also used as benchmarks by the BDDBDDB system [13], one of the most efficient deductive database engine, based on binary decision diagrams (BDDs), that scales to very large Java programs. The benchmarks are all real applications with tens of thousands of users each. Projects vary in the number of classes, methods, bytecodes, variables, and heap allocations. The figures shown in Table 1 are calculated on the basis of a context-insensitive callgraph precomputed by the JOEQ compiler.

Table 2
Times (in seconds) and peak memory usages (in megabytes) for each benchmark and context-insensitive pointer analysis.

Name	time (sec.)	memory (Mb.)
freets	10	61
nfcchat	8	59
jetty	73	70
joone	4	58

All experiments were conducted using Java JRE 1.5, JOEQ version 20030812, on a Intel Core 2 T5500 1.66GHz with 3 Gigabytes of RAM, running Linux Kubuntu 8.04. The analysis times and memory usages of our context insensitive pointer analysis, shown on Table 2, illustrate the scalability of our BES resolution on real examples. Actually, DATALOG_SOLVE solves the (default) query for all benchmarks in a few seconds. The analysis results were verified by comparing them with the solutions computed by the BDDBDDB system on the same benchmark of Java programs and analysis.

5 Conclusion and Future Work

This work described a practical Java program analysis framework that relies on checking rule-based specifications by using a general purpose verification engine. The system, called DATALOG_SOLVE, uses Datalog for encoding complicated program analyses in a few lines, and BES resolution for evaluating the Datalog rules. The tool architecture is based on the well-established verification framework CADP, which provides a generic library for local BES resolution with linear-time complexity. The system is publicly available on http://www.dsic.upv.es/users/elp/datalog_solve.

We plan to optimize DATALOG_SOLVE with further improvements, such as rewriting the Datalog rules in order to support goal-directed bottom-up evaluation, as in the *Magic sets* approach. We also plan to endow our solver with the capability to deal with Java programs containing reflection, by using the rewriting logic framework implemented in the functional programming language Maude [5].

References

[1] Abiteboul, S., R. Hull and V. Vianu, "Foundations of Databases," Addison Wesley, 1995.

[2] Alpuente, M., M. Feliú, C. Joubert and A. Villanueva, *Using Datalog and Boolean Equation Systems for Program Analysis*, in: *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, Lecture Notes in Computer Science **to appear**, 2009.

[3] Andersen, H. R., *Model checking and boolean graphs*, Theoretical Computer Science **126** (1994), pp. 3–30.

[4] Ceri, S., G. Gottlob and L. Tanca, "Logic Programming and Databases," Springer, 1990.

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C.Talcott, "All About Maude: A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic," Lecture Notes in Computer Science **4350**, Springer-Verlag, 2007.

[6] Garavel, H., R. Mateescu, F. Lang and W. Serwe, *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes*, in: *Proceedings of the 19th International Conference on Computer Aided Verification CAV'07*, Lecture Notes in Computer Science **4590** (2007), pp. 158–163.

[7] Mateescu, R., *Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus*, in: *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98*, 1998.

[8] Mateescu, R., *Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems*, Springer International Journal on Software Tools for Technology Transfer (STTT) **8** (2006), pp. 37–56.

[9] Reps, T. W., *Solving Demand Versions of Interprocedural Analysis Problems*, in: *Proceedings of the 5th International Conference on Compiler Construction CC'94*, Lecture Notes in Computer Science **786** (1994), pp. 389–403.

[10] Ullman, J. D., "Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies," Computer Science Press, 1989.

[11] Vieille, L., *Recursive Axioms in Deductive Databases: The Query/Subquery Approach*, in: *Proceedings of the 1st International Conference on Expert Database Systems EDS'86*, 1986, pp. 253–267.

[12] Whaley, J., *Joeq: a Virtual Machine and Compiler Infrastructure*, in: *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators IVME'03* (2003), pp. 58–66.

[13] Whaley, J., D. Avots, M. Carbin and M. S. Lam, *Using Datalog with Binary Decision Diagrams for Program Analysis*, in: *Proceedings of the Third Asian Symposium on Programming Languages and Systems APLAS'05*, Lecture Notes in Computer Science **3780** (2005), pp. 97–118.

[14] Zheng, X. and R. Rugina, *Demand-driven alias analysis for C*, in: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'08* (2008), pp. 197–208.