# Datalog-based Program Analysis with BES and RWL[⋆]

M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,mfeliu,joubert,villanue}@dsic.upv.es

**Abstract.** This paper describes two techniques for Datalog query evaluation and their application to object-oriented program analysis. The first technique transforms Datalog programs into an implicit Boolean Equation System (Bes) that can then be solved by using linear-time complexity algorithms that are available in existing, general purpose verification toolboxes such as Cadp. In order to improve scalability and to enable analyses involving advanced meta-programming features, we develop a second methodology that transforms Datalog programs into rewriting logic (Rwl) theories. This method takes advantage of the preeminent features and facilities that are available within the high-performance system Maude, which provides a very efficient implementation of Rwl. We provide evidence of the practicality of both approaches by reporting on some experiments with a number of real-world Datalog-based analyses.

## 1 Introduction

Datalog [32] is a simple relational query language that allows complex interprocedural program analyses involving dynamically created objects to be described in an intuitive way. The main advantage of formulating data-flow analyses in Datalog is that analyses that traditionally take hundreds of lines of code can be expressed in a few lines [35]. In real-world problems, the Datalog clauses that encode a particular analysis must generally be solved under the huge set of Datalog facts that are automatically extracted from the analyzed program.

We propose two different Datalog query answering techniques that are specially-tailored to object-oriented program analysis. Our techniques essentially consist of transforming the original Datalog program into a suitable set of rules which are then executed under an optimized top-down strategy that caches and reuses "rewrites" in the target language. We use two different formalisms for transforming any given set of definite Datalog clauses into an efficient implementation, namely Boolean Equation Systems (Bes) [5] and Rewriting Logic

[⋆] This work has been partially supported by the EU (FEDER), the Spanish MEC/MICINN under grant TIN 2010-21062-C02-02, and the Generalitat Valenciana under grant Emergentes GV/2009/024. M. A. Feliú was partially supported by the Spanish MEC FPU grant AP2008-00608.

(Rwl) [26], a very general *logical* and *semantical framework* that is efficiently implemented in the high-level executable specification language Maude [9]. This paper provides a comprehensive overview of both techniques, which are fully automatable. For a detailed description of the methods, see [3, 4].

In the Bes-based program analysis methodology, the Datalog clauses that encode a particular analysis, together with a set of Datalog facts that are automatically extracted from program source code, are dynamically transformed into a Bes whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach allows us to reuse existing general purpose analysis and verification toolboxes such as Cadp, which provides local Bes resolution with linear-time complexity. Similarly to the *Query/Subquery* technique [33], computation proceeds with a set of tuples at a time. This can be a great advantage for large datasets since it makes disk access more efficient.

Our motivation for developing our second, Rwl-based query answering technique for Datalog was to provide purely declarative yet efficient program analyses that overcome the difficulty of handling meta-programming features such as reflection in traditional analysis frameworks [22]. Tracking reflective method invocations requires not just tracking object references through variables but actually tracking method values and method name strings. The interaction of static analysis with meta-programming frameworks is non-trivial, and analysis tools risk losing correctness and completeness, particularly when reflective calls are improperly interpreted during the computation. By transforming Datalog programs into Maude programs, we take advantage of the flexibility and versatility of Maude in order to achieve meta-programming capabilities, and we make significant progress towards scalability without losing the declarative nature of specifying complex program analyses in Datalog. The current version of Maude can do more than 3 million rewritings per second on standard PCs, so it can be used as an implementation language [30]. Also, as a means to scale up towards handling real programs, we wanted to determine to what extent Maude is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of Java programs. After exploring the impact of different implementation choices (equations *vs* rules, unraveling *vs* conditional term rewriting systems, explicit *vs* implicit consistency check, etc.) in our working scenario (i.e., sets of hundreds of facts and a few clauses that encode the analysis), we elaborate on an equation-based transformation that leads to efficient transformed Maude-programs.

## Datalog-based program analysis

Datalog is a logic programming language like Prolog, but is does not have data structures such as lists [14]. As a consequence, all queries in Datalog can be guaranteed to terminate, and they have a very simple semantics, enabling aggresive optimizations.

The Datalog approach to static program analysis [35] can be summarized as follows. Each program element, namely variables, types, and code locations are grouped in their respective *domains*. Thus, each argument of a predicate symbol is typed by a domain of values. Each program statement is decomposed into *basic program operations* such as load, store, and assignment operations. Each kind of basic operation is described by a relation in a Datalog program. By considering only finite program domains, and applying standard loop-checking techniques, Datalog program execution is ensured to terminate.

In order to describe the general transformations from Datalog programs into Bes (resp. Maude programs), let us introduce our running example: a context-insensitive points-to analysis borrowed from [35].

*Example 1.* The upper left side of Fig. 1 shows a simple Java program where `o1` and `o2` are heap allocations (extracted by a Java compiler from corresponding bytecode). The Datalog pointer analysis approach consists in first extracting Datalog facts (relations at the upper right side of the figure) from the program. For instance, the relation `vP0` represents the direct points-to information of a program, i.e., `vP0(v,h)` holds if there exists a direct assignment of heap object reference `h` to program variable `v`. Other Datalog relations such as `store`, `load` and `assign` relations are inferred similarly from the code. Using these extracted

```
public A foo { ... p = new Object(); /* o1 */      vP0(p,o1).
                   q = new Object(); /* o2 */      vP0(q,o2).
                   p.f = q;                        store(p,f,q).
                   r = p.f; ... }                  load(p,f,r).
```

```
    vP(V1,H1)      :- vP0(V1,H1).
    vP(V1,H1)      :- assign(V1,V2), vP(V2,H1).
    hP(H1,F,H2)    :- store(V1,F,V2), vP(V1,H1), vP(V2,H2).
    vP(V1,H1)      :- load(V2,F,V1), vP(V2,H2), hP(H2,F,H1).
```

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

facts, the analysis deduces further pointer-related information, like points-to relations from local variables and method parameters to heap objects (`vP(V1,H1)` in Fig. 1) as well as points-to relations between heap objects through field identifiers (`hP(H1,F,H2)` in Fig. 1).

A Datalog query consists of a goal over the relations defined in the Datalog program, *e.g.*, `:- vP(X,Y)`. This goal aims at computing the complete set of program variables in the domain of `X` that may point to any heap object `Y` during program execution. In the example above, the query computes the following answers: {`X/p,Y/o1`}, {`X/q,Y/o2`}, and {`X/r,Y/o2`}.

In the related literature, the solution for a Datalog query is classically constructed following a bottom-up approach; therefore, the information in the query is not exploited until the model has been built [19]. In contrast, the typical top-down, logic programming interpreter would produce the output by reasoning backwards from the query. Between these two extremes, there is a whole spectrum of evaluation strategies [6, 7, 33]. In this work, we essentially consider the top-down approach for developing our techniques since it is closer to Bes local resolution as well as to Maude's evaluation principle, which is based on (non-deterministic) rewriting.

*Related Work.* The description of data-flow analyses as a database query was pioneered by Ullman [32] and Reps [29], who applied Datalog's bottom-up magic-set implementation to automatically derive a *local* implementation.

Recently, Bess with typed parameters [24], called Pbes, have been successfully used to encode several hard verification problems such as the first-order value-based modal $\mu$-calculus model-checking problem [25], and the equivalence checking of various bisimulations [8] on (possibly infinite) labeled transition systems. However, Pbess were not used to compute complex interprocedural program analyses involving dynamically created objects until our work in [3]. The work that is most closely related to the Bes-based analysis approach of ours is [20], where Dependency Graphs (Dgs) are used to represent satisfaction problems, including propositional Horn Clauses satisfaction and Bes resolution. A linear time algorithm for propositional Horn Clause satisfiability is described in terms of the least solution of a Dg equation system. This corresponds to an alternation-free Bes, which can only deal with propositional logic problems. The extension of Liu and Smolka's work [20] to Datalog query evaluation is not straightforward. This is testified by the encoding of data-based temporal logics in equation systems with parameters in [25], where each boolean variable may depend on multiple data terms. Dgs are not sufficiently expressive to represent such data dependencies on each vertex. Hence, it is necessary to work at a higher level, on the Pbes representation.

The idea of using a tabled implementation of Prolog for the purpose of program analysis is a recurring theme in the logic programming community [16]. Oege de Moor *et al.* [16] have developed fast Datalog evaluators that are implemented via optimizing compilation to Sql which performs a specialized version of the well-known 'magic sets' transformation. The system, named *CodeQuest* is specifically suited for source code querying. *CodeQuest* consists of two parts: an implementation of Datalog on top of a relational database management system (Rdbms), and an Eclipse (`www.eclipse.org`) plugin for querying Java code via the Datalog implementation. Datalog queries are compiled in Sql and evaluated by the database system. The database is updated incrementally as the source code changes. Typical queries in *CodeQuest* refer to the enforcement of general rules such as the correct usage of Apis and coding style rules (*e.g.*, declarations and naming conventions), or framework-specific rules (*e.g.*, identify which classes have a method with a given name). Other queries aim to compute metrics or to program understanding (*e.g.*, analyse which methods implement a given abstract method or are never called transitively from the *main* method). The use of a database system as the backend, together with its powerful Rdbms optimizations, makes the evaluation mechanism of *CodeQuest* very scalable. A commercial version has been implemented on top of this work by Semmle [11]. It offers a complete code analysis environment, that stores Java projects as relational databases, and provides an object-oriented query language, called .QL, to allow SQL-like queries on the databases. First, .QL is translated into a pure Datalog intermediate representation, that is then optimised and translated to SQL. Finally, the SQL program can be executed on a number of databases such as Microsoft SQL Server, PostgreSQL and H2. Apart from the completely different evaluation mechanisms and implementation technology, the main differences of our tools, DATALOG_SOLVE and DATALAUDE, with respect to *CodeQuest* is in their focus. While *CodeQuest* focuses on source code queries during the development process, we are more interested in performing dataflow analysis (particularly points-to analysis), that require deeper semantic analysis.

A very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) is available in the BDDBDDB system [35], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach. The computation is achieved by a fixpoint 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 the solutions that satisfy each relation of a Datalog program are exhaustively computed. These sets of solutions are then used to answer complex formulas. In contrast, our approach focuses on demand-driven techniques to solve the considered query with no *a priori* computation of the derivable atoms. 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 are particularly important for program analysis applications. Recently, Zheng and Rugina [36] showed that demand-driven CFL-reachability with worklist algorithm compares favorably with an exhaustive solution. Our technique to solve Datalog programs based on local BES resolution goes in the same direction and provides a novel approach to demand-driven program analyses almost for free.

As for the RWL-based approach, it essentially consists of a suitable transformation from Datalog into Maude. Since the operational principles of logic programming (*resolution*) and functional programming (*term rewriting*) share some similarities [17], many proposals exist for transforming logic programs into term rewriting systems [23, 28, 31]. These transformations aim at reusing the term rewriting infrastructure to run the (transformed) logic program while preserving the intended observable behavior (*e.g.*, termination, success set, computed answers, etc.) Traditionally, translations of logic programs into functional programs are based on imposing an input/output relation (mode) on the parameters of the original program [28]. However, one distinguished feature of Datalog programs that burdens the transformation is that predicate arguments are not *moded*, meaning that they can be used both as input or output parameters. One recent transformation that does not impose modes on parameters was presented in [31]. The authors defined a transformation from definite logic programs into (infinitary) term rewriting for the termination analysis of logic programs. Contrary to our approach, the transformation of [31] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [31] does not tackle the problem of efficiently encoding logic (Datalog) programs containing a huge amount of facts in a rewriting-based infrastructure such as Maude.

*Plan of the Paper.* The rest of the paper is organized as follows: Section 2 describes the application of Datalog and BES to program analysis and reports on experimental results for a context-insensitive pointer analysis of realistic Java programs. Section 3 describes the RWL-based analysis technique and the analysis infrastructure that we deployed to effectively deal with reflection. Finally, Section 4 concludes and discusses some lines for future work.

## 2 The BES-based Datalog evaluation approach

This section summarizes how Datalog queries can be solved by means of Boolean Equation System [5] (BES) resolution. The key idea of our approach is to translate the Datalog specification representing a specific analysis into an implicit BES, whose resolution corresponds to the execution of the analysis [3]. We implemented this technique in the Datalog solver DATALOG_SOLVE that is based on the well-established verification toolbox CADP, which provides a generic library for local BES resolution.

A Boolean Equation System is a set of equations defining boolean variables that can be resolved with linear-time complexity. Parameterised Boolean Equation System [24] (PBES) are defined as BES with typed parameters. Since PBES are a more compact representation than BESs for a system, we first present an elegant and natural intermediate representation of a Datalog program as a PBES. In [3], we established a precise correspondence between Datalog query evaluation and PBES resolution, which is formalized as a linear-time transformation from Datalog to PBES, and vice-versa. As in [35], we assume that Datalog programs have stratified negation (no recursion through negation) and totally-ordered finite domains.

### 2.1 From Datalog to BES

In the following, we illustrate how a PBES can be obtained from a Datalog program in an automatic way. In Fig. 2 we introduce a simplified version of the analysis given in Fig. 1 that contains four facts and the first two clauses that define the predicate `vP`:

Given the query `:- vP(V,o2).`, our transformation constructs the PBES shown below, which defines the boolean variable $x_0$ and three parameterised boolean variables ($x_{vP0}$, $x_{assign}$ and $x_{vP}$), one for each Datalog relation in the analysis. Parameters of these boolean variables are defined on a specific domain and may be either variables or constants. The domains in the example are the heap domain ($D_h = \{o1, o2\}$) and the source program variable domain ($D_v = \{p, q, r, w\}$).

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

**Fig. 2.** Datalog partial context-insensitive points-to analysis

PBES are evaluated by a least fixpoint computation ($\mu$) that sets the boolean variable $x_0$ to *true* if there exists a value for $V$ that makes the parameterised boolean variable $x_{vP}(V, o2)$ true. Logical connectives are interpreted as usual.

$$\begin{array}{ll}
x_0 & \stackrel{\mu}{=} \exists V \in D_v \,.\, x_{vP}(V, o2) \\
x_{vP0}(p, o1) & \stackrel{\mu}{=} \mathsf{true} \\
x_{vP0}(q, o2) & \stackrel{\mu}{=} \mathsf{true} \\
x_{assign}(r, q) & \stackrel{\mu}{=} \mathsf{true} \\
x_{assign}(w, r) & \stackrel{\mu}{=} \mathsf{true} \\
x_{vP}(V\colon D_v, H\colon D_h) & \stackrel{\mu}{=} x_{vP0}(V, H) \vee \exists V2 \in D_v.(x_{assign}(V, V2) \wedge x_{vP}(V2, H))
\end{array}$$

Intuitively, the Datalog query is transformed into the *relevant* boolean variable $x_0$, i.e., the boolean variable that will guide the PBES resolution. Each Datalog fact is transformed into an *instantiated* parameterised boolean variable (no variables appear in the parameters), whereas each predicate symbol defined by Datalog clauses (different from facts) is transformed into a parameterised boolean variable (in the example $x_{vP}(V : D_v, H : D_h)$). This parameterised boolean variable is defined by the disjunction of the corresponding Datalog clauses' bodies, in terms of boolean variables and variable quantifications. Variables that do not appear in the parameters of the boolean variable are existentially quantified on the specific domain (in the example $\exists V \in D_v$ and $\exists V2 \in D_v$).

**From PBES to BES.** Among the different known techniques for solving a PBES (see [10] and the references therein), we consider the resolution method based on transforming the PBES into an alternation-free parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms when data domains are finite [24].

The first step towards the resolution of the analysis is to write the PBES in a simpler format, where, by using new auxiliary boolean variables, each formula at the right-hand side of a boolean equation contains at most one operator. Hence, boolean formulae are restricted to pure disjunctive or conjunctive formulae.

Thereafter, by applying the instantiation algorithm of Mateescu [24], we obtain a parameterless BES where all possible values of each typed data term are enumerated over their corresponding finite data domains. Actually, we do not explicitly construct the parameterless BES. Instead, an implicit representation of the instantiated BES is defined. The interested reader will find the implicit representation in [3]. This implicit representation is then used by the CADP toolbox to generate the explicit parameterless BES on-the-fly. Intuitively, the construction of the BES can be seen as the resolution of the analysis.

However, the idea of naïvely instantiating all the boolean variable parameters in the parameterised BES results in an inefficient implementation since a huge number of possible instantiations are enumerated at each computation step. In order to avoid this, we derive and subsequently optimize a version that instantiates only the parameters necessary to resume the computation. Similarly to *Query/Subquery* [33], we consider the binding of variables occurring in different atoms when transforming a clause: boolean equations only instantiate parameters to the values of variable arguments that appear more than once in the body of the corresponding Datalog clause; otherwise, arguments are kept unbound. In this way, instantiation takes place only when values are needed. Moreover, if the corresponding predicate symbol is extensively defined by a set of facts, the only possible values of its variable arguments in the instantiation are those in the defining facts.

To illustrate the idea behind this optimized version of the generated BES, in Fig. 3 we show (a part of) the BES that results from our running example. Boolean variables, whose name starts with $x$ (shown in bold in the figure) are those that correspond to the goal and subgoals of the original program. Boolean variables starting with $r$ or $g$ are auxiliary boolean variables that are defined during unfolding and instantiation of (sub)goals. The first fragment of the BES (four equations) shows the definition process for the initial query, represented by the boolean variable $x_0$. The query is unfolded and partially instantiated. In our example, there is only one query (`:- vP(V,o2).`) with one single subgoal. Since no variables are shared, $V$ is kept unchanged. Then, the partially instantiated (sub)query is solved by means of its associated boolean variable ($x_{vP(V,o2)}$). Finally, $x_{vP(V,o2)}$ is defined as the disjunction of the boolean variables that correspond to querying the *facts* ($x^f$) and querying the *clauses* ($x^c$).

A query to the clauses of a predicate is defined as the disjunction of the boolean variables that represent the body of the Datalog clauses. In the case of the query $vP(V, H)$ defined by two clauses, the corresponding boolean variable $x^c_{vP(V,H)}$ is defined in terms of two boolean variables $r_{vP0(V,H)}$ and $r_{assign(V,V2),vP(V2,H)}$. The $r$ boolean variable is defined as the disjunction of the different possible instantiations of the query on the shared variables. These *partial instantiations* are represented by $r^{pi}$ boolean variables. For instance, we can observed that $r_{assign(V,V2),vP(V2,H)}$ can be instantiated with the two possible values for $V2$, the only shared variable. The $r^{pi}$ boolean variables are defined as the conjunction of the (partially instantiated) subqueries, which are represented by $x$ boolean variables. As before, boolean variables $x$ are defined as the disjunction of the boolean variables that correspond to querying the *facts* ($x^f$) and querying the *clauses* ($x^c$), as shown in the equation for $x_{assign(V,q)}$. Finally,

$$\begin{aligned}
\boldsymbol{x_0} &\stackrel{\mu}{=} g_{vP(V,o2)} && \textit{unfolding and partial instantiation (pi)}\\
g_{vp(V,o2)} &\stackrel{\mu}{=} g^{pi}_{vP(V,o2)} \\
g^{pi}_{vP(V,o2)} &\stackrel{\mu}{=} \boldsymbol{x_{vP(V,o2)}} && \longrightarrow \textit{partially instantiated query}\\
\boldsymbol{x_{vP(V,o2)}} &\stackrel{\mu}{=} x^f_{vP(V,o2)} \vee x^c_{vP(V,o2)} && \longrightarrow \textit{querying the facts (f) and the clauses (c)}\\
&\vdots\\
x^c_{vP(V,H)} &\stackrel{\mu}{=} \underbrace{r_{vP0(V,H)}}_{clause\ 1} \vee \underbrace{r_{assign(V,V2),vP(V2,H)}}_{clause\ 2}\\
\boxed{r_{vP0(V,H)}} &\stackrel{\mu}{=} r^{pi}_{vP0(V,H)}\\
&\vdots\\
\boxed{r_{assign(V,V2),vP(V2,H)}} &\stackrel{\mu}{=} r^{pi}_{\overset{instantiation}{assign(V,q),vP(q,H)}} \vee r^{pi}_{\overset{instantiation}{assign(V,r),vP(r,H)}}\\
r^{pi}_{assign(V,q),vP(q,H)} &\stackrel{\mu}{=} \boldsymbol{x_{assign(V,q)}} \wedge \boldsymbol{x_{vP(q,H)}} && \longrightarrow \textit{partially instantiated query}\\
\boldsymbol{x_{assign(V,q)}} &\stackrel{\mu}{=} x^f_{assign(V,q)} \vee x^c_{assign(V,q)} && \longrightarrow \textit{querying the facts and the clauses}\\
x^f_{assign(V,q)} &\stackrel{\mu}{=} x^i_{assign(r,q)}\\
x^i_{assign(r,q)} &\stackrel{\mu}{=} \mathsf{true}\\
&\vdots
\end{aligned}$$

**Fig. 3.** An excerpt of the generated Bes.

facts are instantiated to final values and are represented as boolean variables $x^i$, set to true.

As stated above, when the $r^{pi}$ boolean variables are generated, only variables that are shared by two or more subgoals in the body of the Datalog program are instantiated, and only values that appear in the corresponding parameters of the program facts are used. In other words, we do not generate spurious boolean variables, such as $r^{pi}_{assign(V,w),vP(w,H)}$, which can never be true.

**Solution extraction.** By considering the optimized parameterless Bes defined above, the query satisfiability problem is reduced to the local resolution of boolean variable $x_0$. The value (true or false) computed for $x_0$ indicates whether or not there exists at least one satisfiable goal. In order to compute all the different solutions of a Datalog query, it is sensible to use a breadth-first search strategy (Bfs) for the resolution of the Bes. Such a strategy forces the resolution of all boolean variables in the Bfs queue that are potential solutions to the query. Query solutions are extracted from all the boolean variables that are reachable from boolean variable $x_0$ following a path of true-valued boolean variables.

### 2.2 The prototype Datalog_Solve

We implemented the Datalog transformation to Bes in a fully automated Datalog solver tool, called Datalog_Solve[1], which was developed within the Cadp

[1] `http://www.dsic.upv.es/users/elp/datalog_solve/`

verification toolbox [15]. Of course, other source languages and problems can be specified in Datalog and solved by our tool as well.

Datalog_Solve takes as input the Datalog facts that are automatically extracted by the Joeq compiler [34] and a Datalog query that consists of the initial goal and the specification for the analysis.

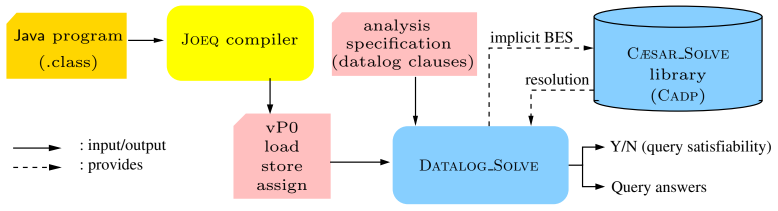

**Fig. 4.** Java program analysis using the Datalog_Solve tool.

The Datalog_Solve architecture (120 lines of Lex, 380 lines of Bison and 3 500 lines of C code) consists of two components, as illustrated in Fig. 4. The front-end of Datalog_Solve constructs the (implicit) optimized Bes representation from the considered Datalog analysis. The back-end of our tool carries out the interpretation of the Bes that is generated and solved on-the-fly by means of the generic Cæsar_Solve library of Cadp.

This architecture clearly separates the implementation of Datalog-based static analyses from the resolution engine, which can be extended and optimized independently.

## 2.3 Experimental Results

In order to test the scalability and applicability of the transformation, the Datalog_Solve tool was applied to a number of Java programs by computing the context-insensitive pointer analysis described in Fig. 1. We have compared our prototype against Bddbddb on four of the most popular 100% Java standalone applications hosted on Sourceforge used as benchmarks for the Bddbddb tool [35]. Execution times (in seconds) are presented in Table 1: "Time" column refers to the analysis computed by our prototype; "Bddbddb" column shows the execution time of the Bddbddb solver; and "Opt.Time" column shows some preliminary results of an ongoing optimization of our prototype, that makes use of an auxiliary data structure (tries) to improve efficiency. This "optimized" approach is still under development [12, 13] and is not fully automated; however, the results are very promising. The results show that our approach works on large amounts of facts as can be encountered in the analysis of real programs. Even with the best encountered boolean variable ordering, the Bdd-based approach appears to be penalized by the poor regularity of the points-to analysis domains and poor redundancy of the analysis relations with respect to our approach based on an explicit encoding.

**Table 1.** Description of the Java projects used as benchmarks.

| Name | Classes | Methods | Vars | Allocs | Time | Bddbddb | Opt.Time |
|---|---|---|---|---|---|---|---|
| freetts (1.2.1) | 215 | 723 | 8K | 3K | 10 | 3.8 | 0.02 |
| nfcchat (1.1.0) | 283 | 993 | 11K | 3K | 8 | 3.86 | 0.01 |
| jetty (6.1.10) | 309 | 1160 | 12K | 3K | 73 | 6.41 | 0.04 |
| joone (2.0.0) | 375 | 1531 | 17K | 4K | 4 | 3.45 | 0.01 |

## 3 The RWL-based Datalog evaluation approach

With the aim to achieve higher expressiveness for static-analysis specification, we translate Datalog into a powerful and highly extensible framework, namely, rewriting logic. Due to the high level of expressiveness of Rwl, many ways for translating Datalog into Rwl can be considered. Because efficiency does matter in the context of Datalog-based program analysis, our proposed transformation is the result of an iterative process that is aimed at optimizing the running time of the transformed program. The basic idea of the translation is to automatically compile Datalog clauses into deterministic equations. Queries and answers are consistently represented as terms so that the query is evaluated by reducing its term representation into a *constraint set* that represents the answers.

### 3.1 From Datalog to Maude

*Membership equational logic* [27] is the subset of Rwl that we use for representing the translated Datalog programs. A *membership equational theory* consists of a signature and a set of equations and membership axioms. Its operational semantics is based on *term rewriting* modulo algebraic axioms, where equations are considered as left-to-right rewriting rules, while membership axioms are assertions of membership to a given sort.

The translated programs have been expressed in Maude [9], which provides many powerful features, like Aci-matching[2], efficient set-representation, meta-programming capabilities (*e.g.*, reflection), and memoization. In this subsection, we first summarize the key ideas of the transformation and its Maude representation, and then we describe how we deal with points-to analyses involving reflection in our framework. The complete transformation is given in [4], and the proof of its correctness and completeness can be found in [2].

**Answer representation.** Datalog answers are expressed as equational *constraints* that relate the variables of the queries to values. Values are represented as *ground terms* of sort `Constant` that are constructed by means of Maude *Quoted Identifiers* (`Qids`). Since logical variables cannot be represented with rewriting

[2] Matching modulo Associativity, Commutativity, and Identity.

rule variables because of their dual input-output nature, we give a representation for them as ground terms of sort `Variable` by means of the overloaded `vrbl` constructor. A `Term` is either a `Constant` or a `Variable`. These elements are represented in Maude as follows:

```
sorts Variable Constant Term .
subsort Variable Constant < Term .
subsort Qid < Constant .
op vrbl : Term -> Variable [ctor] .
```

In our formulation, answers are recorded within the term that represents the ongoing partial computation of the Maude program. Thus, we represent a (partial) answer for the original Datalog query as a set of equational constraints (called answer constraints) that represent the substitution of (logical) variables by (logical) constants that are incrementally computed during the program execution. We define the sort `Constraint` as the composition of answer equations. Elements of sort `Constraint` represent single answers for a Datalog query as follows:

```
sort Constraint .

op _=_ : Term Constant -> Constraint .
op T : -> Constraint .
op F : -> Constraint .
op _,_ : Constraint Constraint -> Constraint [assoc comm id: T] .

eq F, C:Constraint = F .                     --- Zero element
```

Constraints are constructed[3] by the conjunction (`_,_`) of solved equations of the form `T:Term = C:Constant`, the *false* constraint `F`, or the *true* constraint `T`. Note that the conjunction operator `_,_` obeys the laws[4] of associativity and commutativity. `T` is defined as the identity of `_,_`, and `F` is used as the zero element.

Unification of expressions is performed by combining the corresponding answer constraints and checking the satisfiability of the compound. Simplification equations are introduced in order to simplify trivial constraints by reducing them to `T`, or to detect inconsistencies (unification failure) so that the whole conjunction can be drastically replaced by `F`, as shown in the following code excerpt:

```
var Cst Cst1 Cst2 : Constant . var V : Variable .
eq (V = Cst)  , (V = Cst)  = (V = Cst) , T . --- Idempotence
eq (V = Cst1) , (V = Cst2) = F [owise] .     --- Unsatisfiability
```

In our setting, a failing computation occurs when a query is reduced to `F`. If a query is reduced to `T`, then the original (ground) query is proven to be satisfiable. On the contrary, if the query is reduced to a set of solved equations, then the

[3] The actual transformation defines a more complex hierarchy of sorts in order to obtain simpler equations and improve performance.

[4] Associativity, commutativity, and identity are easily expressed by using ACI attributes in Maude, thus simplifying the equational specification and also achieving a more efficient implementation.

computed answer is given by a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ that is expressed by the computed normal form $x_1$ = $t_1$ , ... , $x_n$ = $t_n$.

Since equations in Maude are run deterministically, all the non-determinism of the original Datalog program has to be embedded into the term under reduction. This means that we need to carry all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and we define a new sort called `ConstraintSet` as follows:

```
sorts ConstraintSet .
subsort Constraint < ConstraintSet .
op _;_ : ConstraintSet ConstraintSet -> ConstraintSet  [assoc comm id: F] .
```

The set of constraints is constructed as the (possibly empty) disjunction `_;_` of accumulated constraints. The disjunction operator `_;_` obeys the laws of associativity and commutativity and is also given the identity element `F`.

Transformed predicates are naturally expressed as functions (with the same arity) whose codomain is the `ConstraintSet` sort. They will be reduced to the set of constraints that represent the satisfiable instantiations of the original query. The transformed predicates of our running example are represented in Maude as follows:

```
op vP vP0 assign : Term Term -> ConstraintSet .
```

In order to incrementally add new constraints throughout the program execution, we define the composition operator `x` for constraint sets as follows:

```
op _x_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc] .
```

The composition operator `x` allows us to combine (partial) solutions of the subgoals in a clause body.

**A glimpse of the transformation.** Let us describe the transformation by evaluating queries in our running example. For instance, by executing the Datalog query `:- vP0(p,Y)` on the program in Fig. 2, we obtain the solution {`Y/o1`}. Here, `vP0` is a predicate defined only by facts, so the answers to the query represent the variable instantiations as given by the existing facts. Thus, we would expect the query's RWL representation `vP0('p, vrbl('Y))` to be reduced to the `ConstraintSet` (with just one constraint) `vrbl('Y) = 'o1`. This is accomplished by representing facts according to the following equation pattern:

```
var T0 T1 : Term .
eq vP0(T0,T1) = (T0 = 'p , T1 = 'o1) ; (T0 = 'q , T1 = 'o2) .
eq assign(T0,T1) = (T0 = 'r , T1 = 'q) ; (T0 = 'w , T1 = 'r) .
```

The right-hand side of the RWL equation that is used to represent the facts that define a given predicate (in the example `vP0` and `assign`) consists of the set of constraints that express the satisfiable instantiations of the original predicate. As can be observed, arguments are propagated to the constraints, thus allowing the already mentioned equational simplification process on the constraints. For this particular case, the reduction proceeds as follows:

```
vP0('p,vrbl('Y))
  → ('p = 'p , vrbl('Y) = 'o1) ; ('p = 'q , vrbl('Y) = 'o2)
  →* (T , vrbl('Y) = 'o1) ; (F , vrbl('Y) = 'o2)
  →* vrbl('Y) = 'o1 ; F
  → vrbl('Y) = 'o1
```

Another example of Datalog query is `:- vP(V,o2)`, whose execution for the leading example delivers the solutions {{`V/q`},{`V/r`},{`V/w`}}. Thus, we expect `vP(vrbl('V),'o2)` to be reduced to the set of constraints `(vrbl('V) = 'q) ; (vrbl('V) = 'r) ; (vrbl('V) = 'w)`. In this case, `vP` is a predicate defined by clauses, so the answers to the query are the disjunction of the answers provided by all the clauses defining it. This is represented in Rwl by introducing auxiliary functions to separately compute the answers for each clause, and the equation to join them is as follows:

```
op vP-clause-1 vP-clause-2 : Term Term -> ConstraintSet .
var X Y : Term .
eq vP(X,Y) = vP-clause-1(X , Y) ; vP-clause-2(X , Y) .
```

In order to compute the answers delivered by a clause, we search for the satisfiable instantiations of its body's subgoals. In our translation, we explore the possible instantiations from the leftmost subgoal to the rightmost one. In order to impose this left-to-right exploration, we create a different (auxiliary) unraveling function for each subgoal. Each of these auxiliary functions computes the partial answer depending on the corresponding and previous subgoals and propagates it to the subsequent unraveling function[5]. Additionally, existential variables that occur only in the body of original Datalog clauses, *e.g.*, `Z`, are introduced by using a ground representation that is parameterised with the corresponding call pattern in order to generate fresh variables (in the example below `vrblZ(X,Y)`).

As shown in the following code excerpt, in our example, the first Datalog clause can be transformed without using unraveling functions. For the second Datalog clause (with two subgoals) only one unraveling function is needed in order to force the early reduction of the first subgoal.

```
op vrblZ : Term Term -> Variable .
op unrav : ConstraintSet TermList -> ConstraintSet .

eq vP-clause-1(X,Y) = vP0(X,Y) .
eq vP-clause-2(X,Y) = unrav( assign(X, vrblZ(X,Y)) , X Y ) .
```

The `unrav` function has two arguments: a `ConstraintSet`, which is the first (reduced) subgoal (the original subgoal `assign(X,Z)` in this case); and the `X Y` call pattern. This function is defined as follows:

```
var Cnt : Constant . var TS : TermList .
var C : Constraint . var CS : ConstraintSet .

eq unrav( ( (vrblZ(X,Y) = Cnt , C) ; CS ) , X Y ) =
    ( vP(Cnt,Y) x (vrblZ(X,Y) = Cnt , C) ) ; unrav( CS , X Y ) .
eq unrav( F , TS ) = F .
```

[5] Conditional equations could also be used to impose left-to-right evaluation, but in practice they suffer from poor performance as our experiments revealed.

The unraveling function (in the example `unrav`) takes a set of partial answers as its first argument. It requires the partial answers to be in solved equation form by pattern matching, thus ensuring the left-to-right execution of the goals. The second argument is the call pattern of the translated clause and serves to reference the introduced existential variables. The propagated call pattern is represented as a `TermList`, that is, a juxtaposition (__ operator) of `Terms`. The two `unrav` equations (recursively) combine each (partial) answer obtained from the first subgoal with every (partial) answer computed from the (instantiated) subsequent subgoal (`vP(Cnt,Y)` in the example).

Consider again the Datalog query `:- vP(V,o2)`. We undertake all possible query reduction by using the equations above. Given the size of the execution trace, we will use the following abbreviations: `V` stands for `vrbl('V)`, `vPci` for `vP-clause-i`, and `Z-T0-T1` for `vrblZ(T0,T1)`.

```
vP(V,'o2 )
  → vPc1(V,'o2) ; vPc2(V,'o2)
  →* vP0(V,'o2) ; unrav( assign(V,Z-V-o2) , V 'o2 )
  →* ((V = 'p , 'o2 = 'o1) ; (V = 'q , 'o2 = 'o2))
     ; unrav( ((V = 'r , Z-V-o2 = 'q) ; (V = 'w , Z-V-o2 = 'r)) , V 'o2 )
  →* (F ; (V = 'q , T)) ; (vP('q,'o2) x (V = 'r , Z-V-o2 = 'q))
     ; unrav( (V = 'w , Z-V-o2 = 'r) , V 'o2 )
  →* (V = 'q) ; ((vPc1('q,'o2) ; vPc2('q,'o2)) x (V = 'r , Z-V-o2 = 'q))
     ; (vP('r,'o2) x (V = 'w , Z-V-o2 = 'r)) ; unrav( F , V 'o2 )
  ...
  →* (V = 'q) ; (V = 'r) ; (V = 'w)
```

**Reflection.** Reflection in Java is a powerful technique that is used when a program needs to examine or modify the runtime behavior of applications running on the Java virtual machine. For example, by using reflection, it is possible to write to object fields and invoke methods that are not known at compile time.

The main difficulty of reflective analysis is that we do not have all the basic information for the points-to analysis at the beginning of the computation. This is because Java methods that handle reflection may generate new basic points-to information. A sound approach for handling Java reflection in Datalog analyses is proposed in [22]. We transform Datalog clauses that specify the reflection analysis into Maude conditional rules in a natural way. Then, the Maude reflection capability is used during the analysis to automatically generate the rules that represent the deduced points-to information and adds them to the program. This is in contrast to [22], which resorts to an external artifact with ad-hoc notation and operational principle.

We have implemented a small prototype which essentially consists of a module at the Maude meta-level that implements a generic infrastructure to deal with reflection. Fig. 5 shows the structure of a typical reflection analysis as it is run in our tool. The static analysis is specified in two object-level modules, a *basic module* and a *reflective module*. These modules can be written in either Maude or Datalog since Datalog analyses are automatically compiled into Maude

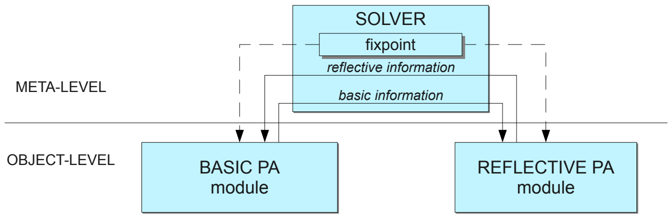

**Fig. 5.** The structure of the reflective analysis.

code. The *basic program analysis* module contains the rules for the classical analysis (which neglects reflection), whereas the *reflective program analysis* module contains the part of the analysis that deals with the reflective components of the Java program. At the meta-level, the *solver* module consists of a generic fixpoint algorithm that feeds the reflective module with the points-to information inferred by the basic analysis. Then, rules that encode the new inferred information are built by the reflective analysis and added to the basic module in order to infer new information, until a fixpoint is reached. A detailed description can be found in [2].

## 3.2 The prototype Datalaude

We implemented the Datalog to Rwl transformation in Datalaude[6] (700 lines of Haskell code and 50 lines of Maude code). The prototype transforms the set of Datalog rules and facts into a Maude membership equational theory. Then, the generated theory is used to reduce each query into its answer constraint set representation in Maude.

**Experimental results.** We report on the performance of Datalaude by comparing it to a previous *rule-based* Datalog-to-Rwl transformation that consisted of a one-to-one mapping from Datalog rules into Maude conditional rules. We briefly present the results obtained by using the rule-based approach and the enhanced *equational-based* Datalaude approach with and without the optimization of using the memoization capability of Maude.

Table 2 shows the resolution times of the three selected versions for different sets of initial Datalog facts (`assign/2` and `vP0/2`), which were extracted by the Joeq compiler [34] from a Java program (with 374 lines of code) that implements a tree visitor algorithm. The evaluated query is `?- vP(Var,Heap)`. Note that the results obtained are progressively better, which emphasizes the fact that by favoring determinism and unconditional equations the computation time can be greatly reduced. Memoization is a table-based mechanism that stores the canonical form (equational simplification) of subterms having operators, at the

[6] `http://www.dsic.upv.es/users/elp/datalaude`

**Table 2.** Number of initial facts (`assign/2` and `vP0/2`) and computed answers (`vP/2`), and resolution time (in seconds) for the three implementations.

| assign/2 | vP0/2 | VP/2 | rule-based | equational | equational+memo |
|---|---|---|---|---|---|
| 100 | 100 | 144 | 6.00 | 0.67 | 0.02 |
| 150 | 150 | 222 | 20.59 | 2.23 | 0.04 |
| 200 | 200 | 297 | 48.48 | 6.11 | 0.10 |
| 403 | 399 | 602 | 382.16 | 77.33 | 0.47 |
| 807 | 1669 | 2042 | 4715.77 | 1098.64 | 3.52 |

top, tagged with the `memo` attribute. Whenever such subterm is encountered during the computation, its canonical form is searched in the table and used instead. Since subcomputations involving the `vP` operator will be repeated many times in the points-to analysis, the overall computation is substantially sped up when the operator `vP` is given the `memo` attribute.

These results confirm that the current DATALAUDE implementation is the one that best fits our program analysis purposes. More details of this experiment and a comparison with other implementations can be found in [4].

## 4 Conclusion and Future Work

This article overviews two novel complementary approaches for solving Datalog queries. Both approaches are fully automatable and applicable to a large class of Datalog programs. In this article, we illustrated them on a popular application domain, namely Datalog-based pointer analysis.

- We used boolean equation systems (BES) to efficiently compute fixpoints in Datalog evaluations. BES resolutions achieve the robustness of bottom-up evaluation, satisfactorily coping with redundant infinite computations. Our transformation also achieves the effectiveness of demand-driven techniques by propagating values and constraints that are part of the query's subgoals in order to speed up the computation.
- We defined, formalized, and proved the correctness of another novel transformation from Datalog programs into Maude programs. By taking advantage of the cogent RWL reflection capabilities, we also demonstrated the adequacy of Maude to support declarative, accurate, and sound complex pointer analyses that include meta-programming features such as reflection in Java programs.

Two new Datalog solvers, called DATALOG_SOLVE and DATALAUDE, respectively, were designed, implemented, and successfully used for the evaluation of the Datalog-based program analysis over several realistic Java programs. The BES-approach is really fast and can analysis in few milliseconds a context-insensitive points-to analysis on a real Java project. Such an approach would perfectly fit in *Integrated Development Environments* (IDEs) to provide rapid feedback to a developer during the development of its code. However, this evaluation technique is not so appropriate to define, compose and experiment new analyses as it would be with a purely declarative approach based on rewriting logic. Rwl offers a sound framework to design complex program analyses in just a few lines.

As ongoing work, we recently endowed Datalog_Solve with new, optimized strategies for local Bes resolution, where Datalog rules are first decomposed in order to allow goal-directed bottom-up evaluation with complexity guarantees [21]. As future work, we plan to explore such sophisticated Datalog optimizations in a purely declarative framework like Maude. Inversely, we could also benefit from the regular structure of our Bes encoding by distributing the Bes resolution over a network of workstations with balanced partitioning while still preserving locality, similarly to [18]. A promising, complementary approach we plan to explore consists in distributing the workload directly at the Datalog level by using Map-Reduce-based algorithms such as [1].

*Acknowledgements.* We are grateful to Fernando Tarín and Adam Kepa for their valuable contributions to the experiments shown in this paper.

## References

1. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In Manolescu, I., Spaccapietra, S., Teubner, J., Kitsuregawa, M., Léger, A., Naumann, F., Ailamaki, A., Özcan, F., eds.: EDBT. Volume 426 of ACM International Conference Proceeding Series., ACM (2010) 99–110
2. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Defining Datalog in Rewriting Logic. Technical Report DSIC-II/07/09, DSIC, Universidad Politécnica de Valencia (2009)
3. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Using Datalog and Boolean Equation Systems for Program Analysis. In Cofer, D., Fantechi, A., eds.: Proc. 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08). Volume 5596 of Lecture Notes in Computer Science., Springer-Verlag (2009) 215–231
4. Alpuente, M., Feliú, M.A., Joubert, C., Villanueva, A.: Defining datalog in rewriting logic. In Schreye, D.D., ed.: LOPSTR. Volume 6037 of Lecture Notes in Computer Science., Springer (2009) 188–204
5. Andersen, H.R.: Model checking and boolean graphs. Theoretical Computer Science **126**(1) (1994) 3–30
6. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems PODS'86, ACM Press (1986) 1–15
7. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (1990)
8. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In: Proc. 18th Int'l Conf. on Concurrency Theory CONCUR'07. Volume 4703 of Lecture Notes in Computer Science., Springer-Verlag (2007) 120–135
9. Clavel, M., Durán, F., Ejer, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Volume 4350 of Lecture Notes in Computer Science. Springer-Verlag (2007)

10. Dam, A., Ploeger, B., Willemse, T.: Instantiation for Parameterised Boolean Equation Systems. In: Proc. 5th Int'l Colloquium on Theoretical Aspects of Computing ICTAC'08. Lecture Notes in Computer Science, Springer-Verlag (2008)
11. de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J.: .QL: Object-oriented queries made easy. In Lämmel, R., Visser, J., Saraiva, J., eds.: GTTSE. Volume 5235 of Lecture Notes in Computer Science., Springer (2007) 78–133
12. Feliú, M., Joubert, C., Tarín, F.: Efficient BES-based Bottom-Up Evaluation of Datalog Programs. In Gulías, V., Silva, J., Villanueva, A., eds.: Proc. X Jornadas sobre Programación y Lenguajes (PROLE'10), Garceta (2010) 165–176
13. Feliú, M., Joubert, C., Tarín, F.: Evaluation strategies for datalog-based points-to analysis. In: Proc. 10th Workshop on Automated Verification of Critical Systems (AVoCS'2010). (2010) To appear
14. Gallaire, H., Minker, J., eds. Advances in Data Base Theory. Plemum Press (1978)
15. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Proc. 19th Int. Conf. on Computer Aided Verification CAV'07. Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (2007) 158–163
16. Hajiyev, E., Verbaere, M., de Moor, O.: *CodeQuest:* Scalable Source Code Queries with Datalog. In: ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings. Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 2–27
17. Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. Journal on Logic Programming **19&20** (1994) 583–628
18. Joubert, C., Mateescu, R.: Distributed On-the-Fly Model Checking and Test Case Generation. In: Proc. 13th Int'l SPIN Workshop on Model Checking of Software SPIN'06. Volume 3925 of Lecture Notes in Computer Science., Springer-Verlag (2006) 126–145
19. Leeuwen, J., ed.: Formal Models and Semantics. Volume B. Elsevier, The MIT Press (1990)
20. Liu, X., Smolka, S.A.: Simple Linear-Time Algorithms for Minimal Fixed Points. In: Proc. 25th Int'l Colloquium on Automata, Languages, and Programming ICALP'98. Volume 1443 of Lecture Notes in Computer Science., Springer-Verlag (1998) 53–66
21. Liu, Y.A., Stoller, S.D.: From datalog rules to efficient programs with time and space guarantees. ACM Trans. Program. Lang. Syst. **31**(6) (2009)
22. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Proc. Third Asian Symposium on Programming Languages and Systems (APLAS'05). (2005) 139–160
23. Marchiori, M.: Logic Programs as Term Rewriting Systems. In: Proc. 4th International Conference on Algebraic and Logic Programming (ALP'94. Volume 850 of Lecture Notes In Computer Science., Springer-Verlag (1994) 223– 241
24. Mateescu, R.: Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In: Proc. 2nd Int'l Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98. (1998)
25. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Proc. 15th Int'l Symp. on Formal Methods FM'08. Volume 5014 of Lecture Notes in Computer Science., Springer-Verlag (2008)
26. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1) (1992) 73–155

27. Meseguer, J.: Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., ed.: WADT. Volume 1376 of Lecture Notes in Computer Science., Springer (1997) 18–61
28. Reddy, U.: Transformation of Logic Programs into Functional Programs. In: Proc. Symposium on Logic Programming (SLP'84), IEEE Computer Society Press (1984) 187–197
29. Reps, T.W.: Solving Demand Versions of Interprocedural Analysis Problems. In: Proc. 5th Int'l Conf. on Compiler Construction CC'94. Volume 786 of Lecture Notes in Computer Science., Springer-Verlag (1994) 389–403
30. Rosu, G., Havelund, K.: Rewriting-Based Techniques for Runtime Verification. Autom. Softw. Eng. **12**(2) (2005) 151–197
31. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Proc. 16th Int'l Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'06). Volume 4407 of Lecture Notes in Computer Science., Springer-Verlag (2007) 177–193
32. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies. Computer Science Press (1989)
33. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Proc. 1st Int'l Conf. on Expert Database Systems EDS'86. (1986) 253–267
34. Whaley, J.: Joeq: a Virtual Machine and Compiler Infrastructure. In: Proc. Workshop on Interpreters, Virtual Machines and Emulators IVME'03, ACM Press (2003) 58–66
35. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Proc. Third Asian Symp. on Programming Languages and Systems APLAS'05. Volume 3780 of Lecture Notes in Computer Science., Springer-Verlag (2005) 97–118
36. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'08, ACM Press (2008) 197–208