A Survey of Algorithmic Debugging

RAFAEL CABALLERO and ADRIÁN RIESCO, Universidad Complutense de Madrid
JOSEP SILVA, Universitat Politècnica de València

Algorithmic debugging is a technique proposed in 1982 by E. Y. Shapiro in the context of logic programming. This survey shows how the initial ideas have been developed to become a widespread debugging schema fitting many different programming paradigms and with applications out of the program debugging field. We describe the general framework and the main issues related to the implementations in different programming paradigms and discuss several proposed improvements and optimizations. We also review the main algorithmic debugger tools that have been implemented so far and compare their features. From this comparison, we elaborate a summary of desirable characteristics that should be considered when implementing future algorithmic debuggers.

Categories and Subject Descriptors: F.3.1 [**Theory of Computation**]: Logics and meaning of programs—*Specifying and verifying and reasoning about programs*; D.3.1 [**Software**]: Programming Languages—*Formal definitions and theory*

General Terms: Languages, Theory

Additional Key Words and Phrases: Algorithmic debugging, declarative debugging, software engineering

ACM Reference format:
Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A Survey of Algorithmic Debugging. *ACM Comput. Surv.* 50, 4, Article 60 (August 2017), 35 pages.
https://doi.org/10.1145/3106740

1 INTRODUCTION

In the 1980s, logic programming was a mature and well-established programming paradigm (Kowalski 2014). The main goal of logic programming languages such as Prolog (Sterling and Shapiro 1986) was, and continues to be, to allow the programmer to focus on *what* the program must do, leaving the system implementation the problem of *how* to achieve this.

Ehud Y. Shapiro proposed to extend this point of view from programming to debugging in his seminal work (Shapiro 1982b), further developed as part of his Ph.D. thesis, the latter being selected as the *1982 ACM Distinguished Dissertation* (Shapiro 1982a). In these works, Shapiro proposed a

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R, TIN2016-76843-C4-1-R, *StrongSoft* (TIN2012-39391-C04-04), and *TRACES* (TIN2015-67522-C3-3-R) by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic) and by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).
Authors' addresses: R. Caballero and A. Riesco, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, C/ Profesor José García Santesmases, 9, E-28040, Madrid, Spain; emails: {rafacr, ariesco}@ucm.es; J. Silva, Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Camino de Vera s/n, E-46022, Valencia, Spain; email: jsilva@dsic.upv.es.

https://doi.org/10.1145/3106740

new debugging technique called *algorithmic debugging*, also known later as *declarative debugging*. Algorithmic debuggers allow the user to focus on the program semantics, comparing the *intended behavior* of the program and the real computations.

In Shapiro's proposal, the algorithmic debugging technique starts when some logic programming goal produces an unexpected result. This unexpected result or *initial symptom* can be either an erroneous substitution (a *wrong answer* in the rest of the article) or even an unexpected finite failure (which we call *missing answer*). Then, the debugger internally repeats the erroneous computation but builds at the same time a tree that reflects the logic of the computation. In this *debugging tree*, the root corresponds to the initial computation. The descendants of every node correspond to the subcomputations that occur in the implementation of the computation stored at the parent node. Each node has associated to it both a result and the fragment of code used to perform the computation that produces this result.

Once the debugging tree has been obtained, the debugger interacts with an external *oracle*, usually the user, checking the validity of some tree nodes. Usually, the question is *"Did you expect this computation to produce this result?"* If the result is expected, then the node is marked as *valid* and otherwise as *invalid*. Notice, however, that an invalid node does not necessarily imply that the fragment of code associated with the node contains a bug. It might occur that the node contains an erroneous result due to the misbehavior of some auxiliary subcomputation, that is, due to the erroneous result at a child node, following the *"garbage in, garbage out"* principle.

Thus, the debugger's goal is to find a computation that has produced an erroneous result from correct subcomputations or, in terms of the debugging tree terminology, an invalid node with only valid children, called simply a *buggy node* in the rest of the article. The debugger reports the fragment of code associated with a buggy node as erroneous to the user and the debugging task ends.

It is worth noticing that algorithmic debuggers usually focus on finding a buggy node in each debugging session. Once the error has been corrected, the user can run the program again and, if a new error occurs, start the debugger once more. Thus, the debugger does not need to ask the user about the correctness of *all* the nodes in the debugging tree, only about those that lead to the location of a buggy node. The algorithm that chooses a node to be asked about is often called *navigation strategy*. A possible *navigation strategy* is to proceed bottom-up, asking the oracle about the validity of leaf nodes first and then about their ancestors. If a leaf node is invalid, then it is buggy (because it has no invalid children), and its associated implementation is pointed out as erroneous. If it is valid, then another node is selected following a post-order traversal of the tree and the process is repeated until a buggy node is found. This strategy was called *single stepping* in Shapiro (1982a), which also suggests a more efficient strategy called *divide-and-query*. Since then, many other strategies have been proposed (Silva 2011).

It soon became apparent that Shapiro's ideas could also be applied to other programming paradigms and that, in fact, they constituted a new and powerful general debugging framework (Naish 1997a). For this reason, in the following years, the technique influenced the development of new debuggers in many programming paradigms, including declarative programming, object-oriented programming, and database query languages. Moreover, the ideas introduced by algorithmic debugging expanded to other areas such as the learning of programming languages.

Nevertheless, the evolution of the technique has not been as successful as its properties seemed to promise. Despite a good number of attempts to produce mature algorithmic debugging tools, the technique still has to make the jump from academia to the industry, and this deserves a critical analysis.

This survey reviews the theoretical and practical advances made in algorithmic debugging since its presentation in 1982. It also discusses the impact of the technique both in debugging and in other fields and its strengths and limitations, and, finally, it proposes possible lines of future evolution.

The next section presents the basis of algorithmic debugging as a general framework. Then, in Section 3, we describe how the general framework has been instantiated in different programming paradigms, discussing the particularities of each case. Section 4 outlines different applications of algorithmic debugging besides debugging. The different issues related to algorithmic debugging are discussed in Section 5. In Section 6, we describe how most of these issues have been faced with different techniques. Together, the sections form a collection of desirable features and characteristics of an algorithmic debugger. In Section 7, we compare the main algorithmic debuggers proposed so far. Finally, Section 8 concludes this survey and proposes future lines of work.

2 THE GENERAL FRAMEWORK

We start by introducing the basic principles of algorithmic debugging as an abstract general debugging framework.

2.1 Framework Components

As sketched in the introduction, algorithmic debugging can be depicted as a two-phase process, which starts when an unexpected result is detected:

- *Phase 1* automatically builds a suitable data structure representing the computation that produces the unexpected result. This structure is usually a tree, which we denominate in this article as *debugging tree*.
- *Phase 2* is often called the *navigation phase*. During this phase the debugger asks an external oracle questions looking for a buggy node. The oracle must determine if the results associated with the nodes are expected, that is, if they match the *intended behavior* of the computation.

In phase 1, each debugging tree node corresponds to a computation step. Suitable computation steps must have:

(1) An associated computation result.
(2) An associated fragment of code or system component, responsible for carrying out the computation and producing the result.
(3) A notion of dependence among (sub)computations.

The definition of these components implicitly define the structure of the *intended interpretation* of a program, which is the set containing all the possible computations with their expected results. A node will be marked as valid when it is part of or is entailed by the intended interpretation and marked as invalid otherwise.

The first attempt of defining a general scheme for algorithmic debugging was defined by Lee Naish (1997a), who also discusses some properties of debugging trees regardless of the underlying language. A presentation of the general schema, following Tamarit et al. (2016) is depicted in Figure 1. The debugger constructs the debugging tree associated to the initial symptom, as required in phase 1. Implicitly, we assume that the tree nodes are labelled with a state, initially *unknown* (the default). At the beginning of the navigation phase, the root of the tree is marked as invalid, since it corresponds to the initial symptom. During the navigation phase, the debugger picks up unknown nodes and marks them according to the oracle's answers until a buggy node is found (the function `buggy(T)` that returns the first invalid node with valid children found or $\perp$ if such node does not exist yet). Finally, the buggy node is returned. As observed in Naish (1997a), the schema returns just the first buggy node, indicating that its associated component (usually a fragment of program code) is incorrect. Of course, it is possible to modify the general setting to obtain all the buggy nodes, but this is usually avoided, because (1) often different buggy

```
debugger(initialSymptom) ->
    T = generate_debugging_tree(initialSymptom)
    mark root(T) as invalid
    while (buggy(T) == ⊥)
          Choose a node N ∈ T with (mark(N) == unknown)
          Ask the oracle about the validity of N
          Mark T accordingly
    return buggy(T)
```

Fig. 1. General framework describing algorithmic debugging.

```
debugger(initialSymptom) ->
    T = generate_debugging_tree(initialSymptom)
    mark root(T) as invalid
    while (|T| ≠ 1)
          Choose a  node N ∈ T marked as unknown
          Ask the oracle about the validity of  N
          If the user indicates that N is valid then
             remove N and its subtree from T
          else
             mark N as invalid
             T = subtree rooted by N
    return root(T)
```

Fig. 2. Improved framework for algorithmic debugging.

nodes may correspond to the same bug, and (2) in most algorithmic debuggers, the oracle is the user, and thus it is important to decrease the number of questions. For these reasons, algorithmic debuggers usually only look for *weak completeness*, which can be enunciated as follows:

Theorem 2.1. *Let T be a debugging tree containing some invalid node. Then, T contains at least one buggy node.*

The proof uses induction on the size of the tree. Correctness, however, is not so simple and requires the conversion of the following assumption into a formal result:

Assumption 2.2. *Let T be a debugging tree and N a buggy node in T. Then, the fragment of code associated with N is incorrect.*

Proving this result is beyond the possibilities of the general setting. Each particular instance must prove, or at least justify in the case of languages with complex or involved semantics, that this assumption holds and thus that the debugger is correct.

Figure 2 shows an alternative to the general setting of Figure 1, where the size of the debugging tree is reduced each time the question about a new node is answered. This is useful for *lazy* algorithmic debuggers that avoid building the whole tree for efficiency reasons. In these debuggers, removing a subtree rooted by a node *N* often means not constructing the subtree associated with *N*, thus saving time and space. Each loop iteration singles out a node N whose state is *unknown* following a navigation strategy (Silva 2011). Then, the debugger asks the oracle about the validity on N. If the oracle indicates that the node N contains an expected result, then N and its subtree is removed from the tree. If, on the contrary, the user indicates that the result at N is unexpected, then N is marked as invalid and its subtree is chosen as the new debugging tree.

Each iteration decreases the size of the tree. Notice also that, at the beginning of every iteration, the tree root is invalid. Then, assuming that we start with a non-empty debugging tree, it is straightforward to prove that the debugger ends with a tree such that ($|T| = 1$). To ensure that this final node is a buggy node, we must check that the two operations

(1) remove a subtree rooted by a valid node and (2) choose any subtree with an invalid root as a debugging tree, are safe. This means that the new debugging tree must contain at least one buggy node and also that all buggy nodes found in the new debugging tree exist and are buggy in the initial tree.

These properties are established by the following proposition:

Proposition 2.3. *Let T be a debugging tree with root R. Suppose that R is marked as invalid. Then:*

(1) Let N be an invalid node in T. Let T' be the subtree rooted by N. Then, every buggy node found in T' is also buggy in T.
(2) Let N be a valid node in T. Let T' be the tree obtained after removing every subtree with root N from T. Then, T' contains at least one buggy node N', and every buggy node N' in T' occurs and is buggy in T as well.

Both properties are easy to prove and, combined with the weak completeness (Theorem 2.1), guarantee that the new framework is still *weakly* complete in the sense that it always finishes pointing out a buggy node. They also guarantee that the correctness result of Assumption 2.2 holds, that is, that the fragment of code pointed out as incorrect is always associated with a buggy node of the original tree.

Many variants have been proposed since the presentation of the general framework. For instance, it is usual to allow for other answers beyond *valid* and *invalid*, such as *don't know/maybe* (Wlodzimierz et al. 1988) to avoid answering difficult questions, or allowing for the *inadmissible* answer (Pereira 1986; Naish 1997b) for calls to subprograms with unexpected parameters (see Section 6.4 for details). Recently, one of the authors of this article has participated in the proposal of an extension of the general schema (Insa and Silva 2015b), but, for the sake of simplicity, we maintain the original schema as presented in Figures 1 and 2.

2.2 Implementation Techniques

The general framework starts representing the computation by means of a suitable debugging tree. This first automatic phase requires repeating the computation that produced the initial symptom. But how is the debugging tree obtained while repeating the computation? Two main alternatives have been proposed:

(1) Using a source-to-source program transformation (Pope 1998; Caballero and Rodríguez-Artalejo 2002; Pope and Naish 2003; Chitil et al. 2003; Pope 2006; Lux 2008). The idea is to transform the program to be debugged, such that the code in charge of each computation returns its associated debugging tree combined with the original result. For instance, a function that returns a value of type T may return a value of type (T, *DebugTree*) after the transformation, with *DebugTree* being a suitable datatype defined to represent the debugging tree.
(2) Using code instrumentation (Nilsson and Fritzson 1992, 1994; Nilsson 1998; Faddegon and Chitil 2015), reflection (Shapiro 1982a; Lloyd 1987b; Binks 1995; Tessier and Ferrand 2000; MacLarty 2005; Riesco et al. 2012; Caballero et al. 2015), or directly modifying the compiler (Nilsson 2001) to produce the debugging tree.

The advantage of the program transformation is that it can be defined with precision regardless of the compiler/interpreter. This is also very convenient for proving properties such as well-typedness of the transformed program or the above-mentioned algorithmic debugging correctness. However, this approach also presents two important disadvantages:

(1) In some cases, as in lazy functional languages, a new primitive, usually called *dVal* or *pVal* (Nilsson and Sparud 1997), needs to be introduced. This primitive produces the representation of a term at the moment where its computation occurs. The semantics of this primitive does not match the standard semantics of these languages, where the execution order is not predetermined. Thus, *dVal* is an *impure* function, which worsens reasoning about the transformed program in a purely declarative context.
(2) The main problem of the program transformation is efficiency. The transformed program is usually much slower and requires much more memory to record the different parts of the debugging tree.

The inherent lack of efficiency in the program transformation approach leads many designers of algorithmic debuggers to either use the reflection capabilities of the language when executing the code, or simply define an *ad hoc* execution mechanism to produce the debugging trees more efficiently.

3 ALGORITHMIC DEBUGGING IN DIFFERENT PARADIGMS

In this section, we review the adaptation of the principles of algorithmic debugging to different programming paradigms.

3.1 Logic Programming

Algorithmic debuggers for programs written in logic languages based on Horn clauses (e.g., Prolog (Lakhotia and Sterling 1991)) usually consider two different types of errors: *wrong answers*, when the user obtains an unexpected computed answer for some initial goal, and *missing answers*, when there is an expected answer that is not covered by the set of computed answers.

A generic algorithmic debugger of logic programs can cover both types of errors simultaneously. In fact, this is desirable in the case of programs allowing for the use of negation, since this feature may convert the missing answer of some predicate into a wrong answer for another predicate and the other way round (Lloyd 1987b). However, it is conceptually useful to consider both types of errors as two different instances of the general setting and this is the point of view considered in this article.

In the case of *wrong answers*, the computations correspond to predicate calls. Then, the instance of the general framework described in Section 2.1 consists of the following components:

(1) The result of a computation is a substitution of logic variables with terms. Note that the same predicate call may allow for many substitutions as solutions, but, in this case, we focus on the substitution employed to obtain the wrong answer.
(2) The fragment of code associated with each node is the predicate clause employed to solve the predicate call producing the result.
(3) Each computation depends on the computation of the predicate calls associated with the atoms occurring in the body of the predicate clause.

If, instead of wrong answers, we consider *missing answers*, then the computations still correspond to predicate calls, but the components vary:

(1) The result of the computation is now the set of all the computed substitutions for the given predicate call.
(2) The fragment of code associated is the whole predicate, including all its clauses.
(3) Each computation depends on the computation of the predicate calls associated with the atoms occurring in the bodies of all the predicate clauses employed during the computation.

To include the initial goal G as the root of the tree, it is usually assumed that the debugger implicitly introduces a new predicate *main* defined by one rule `main(`$X_1, \ldots, X_n$`) :- G.`, where $X_1, \ldots, X_n$ are the free variables in G.

The intended interpretation, valid for both wrong and missing answers, can be considered a set of ground atoms (that is, atoms containing no variables). If the program is correct, then the set of answers $Sol(p(t_1, \ldots, t_n)) = \{\theta_1, \ldots, \theta_n\}$ for a given predicate call $p(t_1, \ldots, t_n)$ exactly covers the set of instances of $p(t_1, \ldots, t_n)$ that are true in the intended interpretation $\mathcal{I}$ of the program. Formally:

Definition 3.1. Let P be a logic program. Let $\mathcal{I}$ be the intended interpretation of P and $p(t_1, \ldots, t_n)$ a predicate call. Let $Sol(p(t_1, \ldots, t_n)) = \{\theta_1, \ldots, \theta_n\}$ be the set of solutions obtained for $p(t_1, \ldots, t_n)$ with respect to P. Then, we say that the set of solutions is expected if, for every substitution θ such that $\theta(p(t_1, \ldots, t_n))$,

$$\theta(p(t_1, \ldots, t_n)) \in \mathcal{I} \text{ iff}$$
$$\text{there exists a substitution } \mu \text{ and an index } i,\ 1 \leq i \leq n,$$
$$\text{such that } \theta(p(t_1, \ldots, t_n)) \equiv \mu(\theta_i(p(t_1, \ldots, t_n))).$$

Otherwise, we say that:

(1) A *wrong answer* occurs when there is some index i, $1 \leq i \leq n$, and some substitution μ such that $\mu(\theta_i(p)) \notin \mathcal{I}$.
(2) A *missing answer* occurs when some substitution θ' exists such that $\theta'(p) \in \mathcal{I}$, but there is no index i, $1 \leq i \leq n$, and some substitution μ such that $\theta'(p) \equiv \mu(\theta_i(p))$.

In the case of logic programs with wrong answers, the algorithmic debugger ends by pointing to an incorrect predicate clause. In the more contrived case of missing answers, a whole predicate is marked as *incomplete* by the debugger. Wrong answers are easier to detect than missing answers, because the former focuses on a particular solution, while the latter requires the whole solution set.

To illustrate these ideas, we use the same example introduced by Shapiro (1982b), which corresponds to the Prolog implementation of the Quicksort algorithm shown in Figure 3. As usual in logic programming, uppercase identifiers represent variable symbols, and the notation `[X|Y]` represents a list with head (first element) `X` and tail (list containing the rest of the elements) `L`. The empty list is represented as `[]`.

The *intended behavior* of the predicates in the program contains ground atoms of the form:

— `qsort(l1,l2)` where `l2` is the result of sorting the list `l1`.
— `partition(l,x,l1,l2)` where `l1` contains those elements of list `l` lower or equal to `x`, while `l2` contains the elements of `l` that are greater than `x`.
— `append(l1,l2,l3)` where `l3` is the result of concatenating the elements on list `l2` at the end of list `l1`.

However, the program contains a bug: The first clause of `partition` lacks the condition $X > Y$. For instance, the goal $p \equiv$ `qsort([2,1],X)` is expected to produce just one answer, the result being

```
qsort([X|L],L0) :-
    partition(L, X, L1, L2),
    qsort(L1, L3), qsort(L2, L4),
    append(L3,[X|L4],L0).
qsort([],[]).

partition([X|L], Y, L1, [X|L2]) :- partition(L,Y,L1,L2).
partition([X|L], Y, [X|L1], L2) :- X <= Y, partition(L,Y,L1,L2).
partition([],X,[],[]).

append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).
append([],L,L).
```

Fig. 3. A logic program for quicksort.

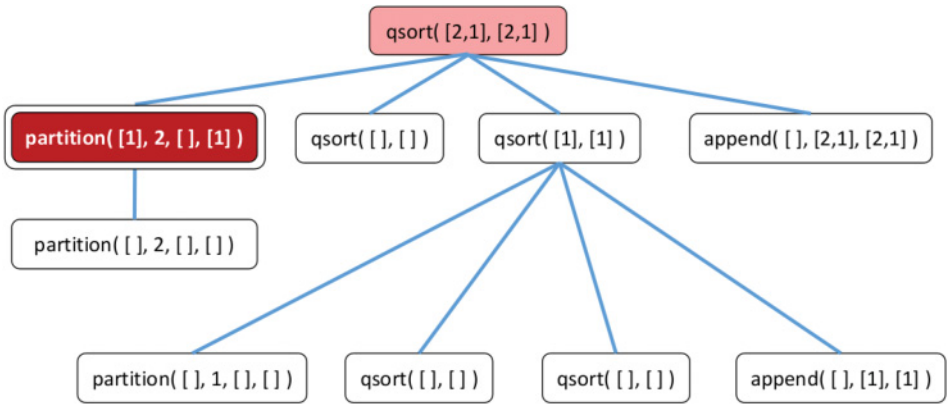

Fig. 4. Proof tree for the goal `qsort([2,1],X)` and its wrong answer `{ X / [2,1] }`.

the substitution `{ X / [1,2] }`. However, the logic system yields just one answer $\theta_1 \equiv$ `{ X / [2,1] }` showing a discrepancy between the intended behavior and the computed result.

Note that this is a case where wrong and missing answers coexist. Obviously, `{ X / [2,1] }` is a wrong answer according to Definition 3.1, because for $i = 1$ and $\mu = id$ (with id being the identity substitution), we have $\mu(\theta_1(p))$ = `qsort([2,1],[2,1])` $\notin \mathcal{I}$. But there is also a missing answer according to the same definition, because when $\theta' \equiv$ `{ X / [1,2] }`, $\theta'(p)$ = `qsort([2,1],[1,2])` $\in \mathcal{I}$, but $i = 1$, the only solution index possible, verifies that there is no possible substitution μ such that `qsort([2,1],[1,2])`$\equiv \mu($`qsort([2,1],[2,1])`$)$. When both errors occur simultaneously, the rule of thumb is to only report the wrong answer, since it produces simpler debugging sessions in general.

Figure 4 shows the debugging tree for the erroneous answer θ_1 applied to the goal p. The two invalid nodes are marked in bold. The double-framed node associated with `partition` is the only buggy node. It is indeed invalid, because splitting the list `[1]` with respect to the pivot `2` should produce `[1]` and `[]`, instead of `[]` and `[1]`. It is also easy to check that its only child `partition([],2,[],[])` is valid, since the only possible split of the empty list, regardless of the pivot, is in two empty lists. Following a top-to-bottom, left-to-right navigation strategy, the debugger finds the buggy node in two questions (the root is invalid by default and does not require any question). The output of the debugger after these two questions would be similar to this:

```
Erroneous clause found.
Clause partition.1: partition([X|L], Y, L1, [X|L2]):- partition(L,Y,L1,L2).
Erroneous instance: partition([1], 2, [], [1]):- partition([],2,[],[]).
```

The error indicates the predicate, the particular clause, and it can also display the instance associated with the buggy node. This last bit of information is really useful for locating the precise error, which in this case is that this clause should only be applied if the first element of the list is greater than the pivot. Therefore, adding `X > Y` at the beginning of the clause body solves the problem.

It is worth noting that in the case of logic programming the debugging tree directly corresponds to the concept of *proof tree*. This is interesting, because this allows us to formally prove the correctness of the technique. The results, as well as a general and detailed explanation of algorithmic debugging in the field of logic programming, can be found in Lloyd (1987b).

Shapiro's original works were soon followed by others, still in the field of logic programming (Av-Ron 1984; Huntbach 1987; Ferrand 1987; Lloyd 1987b; Naish et al. 1989; Shmueli and Tsur 1991; Naish 1992b) and extending to constraint logic programming (CLP). In almost all cases, the debugging tree is obtained by using a meta-interpreter, taking advantage of the reflection possibilities of logic languages such as Prolog, and all the works include results of soundness and completeness. A very similar approach was later applied to the more general CLP scheme (Fromherz 1993; Tessier and Ferrand 2000), to the combination with assertions (Wlodzimierz et al. 1988), and to multi-paradigm languages such as Gödel or Maude (Binks 1995; Riesco et al. 2012).

Deductive Databases (Ramakrishnan and Ullman 1995) are databases based on the logic programming paradigm. Its more conspicuous representative is Datalog (Ceri et al. 1989), whose main differences with the logic programming language Prolog are as follows:

(1) In Datalog, programs are defined by Horn clauses, as in Prolog, but only constant variables and constant symbols are allowed.
(2) Programs can contain only a simplified form of negation called *stratified negation* (Apt et al. 1988).
(3) Programs are evaluated bottom-up. Given an initial goal, a Datalog system returns a set with all the ground solutions satisfying the goal with respect to the program.

The first two points ensure program termination, a property usually required by database query languages, while the third ensures that all the database facts that correspond with a query (the name of goals in this context) are returned.

Given its similarity with Prolog, it is natural that algorithmic debugging was proposed as the suitable way of debugging Datalog programs. The first work considering the algorithmic debugging of programs in Datalog is Russo and Sancassani (1992). This article proposes to use a variant of the Selective Linear Definite (SLD) (Kowalski and Kuehner 1971) clause resolution rule to find the errors. This initial approach considers as many debugging trees as atomic answers the query produces, which is not realistic in queries over large databases, which can potentially return numerous atomic results. Other authors (Arora et al. 1993; Wieland 1990) proposed to use forests of debugging trees.

However, trying to represent Datalog programs using Prolog semantics does not account for a new type of error, which was not present in logic programs: the *uncovered sets of atoms*, caused by *incomplete sets of relations*. Consider the following Datalog program from Caballero et al. (2008):

```
p(V)  :-  q(V).
q(V)  :-  p(V).
```

Suppose that the intended interpretation is `{p(a), q(a)}`, and that we try the query `p(V)`. In Prolog, this goal does not terminate, but in a Datalog system such as DES (Sáenz-Pérez 2011), it

returns the empty set {}. In the theory of algorithmic debugging for logic programming, this is considered a missing answer, and the instance `p(a)` a missing instance, which implies the existence of an incomplete predicate definition. But this is not the case here, since the relation `p` and `q` can produce the values `p(a)`, `q(a)` using the instance $\theta = \{\mathtt{V} \mapsto \mathtt{a}\}$. Instead, we say that the set `{p(a),q(a)}` form an *uncovered set of atoms*, while the set $S = \{\mathtt{p},\mathtt{q}\}$ determines an incomplete set of relations. A curious consequence of this type of error is that debugging trees are not the suitable structure for representing Datalog computations. Instead, *debugging graphs* are the suitable structure (Caballero et al. 2008; Köhler et al. 2012). In this new setting, the concept of buggy node remains, but it is enriched with the new concept of *buggy circuit*, a graph circuit formed by invalid nodes such that all the nodes directly connected to those of the circuit are valid. It has been proved (Caballero et al. 2008) that a buggy circuit either contains a node corresponding to an incorrect program rule or determines an incomplete set of predicates.

Similar ideas have been applied to debug SQL queries (Caballero et al. 2012), taking advantage of the declarative nature of this query language.

3.2 Functional Programming

Ten years after E. Y. Shapiro's initial proposal, H. Nilsson and P. Fritzson (1992, 1994) and, independently, L. Naish (1992a) proposed to apply algorithmic debugging to another main declarative programming paradigm stream: functional programming.

Soon, an interesting difference with the logic paradigm was noticed: missing answers, the most complicated source of errors in logic programming, can be considered as just a particular case of wrong answers in the case of functional programming. This is accomplished by simply assuming that a failing computation returns a special value *error* and generating the corresponding debugging tree. It is worth observing that this simple idea cannot be applied to logic languages due to their more involved execution search mechanism, which produces much more complex trees (Naish 1992b). In contrast, functional languages assume that any function call will succeed for (at most) one function rule, usually the first rule in textual order whose arguments match the calling parameters, and no backtracking mechanism is involved in the process.

The algorithmic debuggers proposed for functional programming adapted the general schema of Figure 1 to functional languages such as Haskell (Hutton 2016) with the following assumptions:

(1) Computing in the functional paradigm means evaluating expressions. The computation result is the result obtained from this evaluation. It is usually assumed that expressions are function calls (the initial expression is implicitly the body of a `main` function).
(2) Thus, each computation is associated with a *function* in the program.
(3) The subcomputations associated with a computation of a given function call correspond to the function calls in the body of the function rule used to evaluate the expression.

However, algorithmic debugging of functional programs introduces two new difficulties:

(1) Apparently, nodes can become very complex due to the existence of nested function calls. This in turn would produce complex questions, difficult to answer. The obvious solution is to evaluate the arguments of function calls in advance, reducing nested calls to values. The evaluation of these nested function calls are considered subcomputations and, consequently, are represented as child nodes.
(2) However, the idea of evaluating the arguments in advance poses a new difficulty in the case of *lazy* functional languages. In these languages, *potentially* infinite structures are allowed, although the actual computations only evaluate the part needed in each case. Thus, the debugger must be careful to evaluate the arguments only to the point required by the particular computation to be debugged.

```
data Nat = Z | S Nat

take Z     _      = []
take _     []     = []
take (S n) (x:xs) = x:take n xs

from n            = n:from n
```

Fig. 5. A funcional program with a datatype and two simple functions.

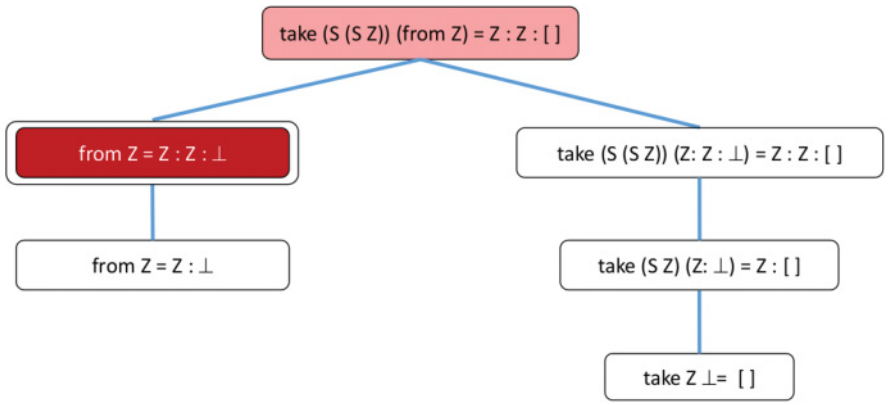

Fig. 6. Evaluation Dependence Tree for the evaluation of the expression `take (S (S Z)) (from Z)` and its associated wrong answer `Z:Z:[]`.

The two difficulties were overcome with the definition of the *Evaluation Dependence Tree* (EDT in short) in Nilsson and Sparud (1997). The EDT abstracts the details about the execution away, only displaying the arguments evaluated to the point required by the computation.

Figure 5 presents a simple example. The program defines a datatype `Nat` representing the natural numbers using the Peano notation. Each number can be either zero (`Z`) or the successor (`S`) of another natural number. The program also uses the built-in datatype *list*, where the empty list is represented as in logic programming (`[]`), while non-empty lists are represented as `(x:xs)` to indicate that `x` is the first element and `xs` the rest of the list. The program also defines the functions `take` and `from` with the following intended behavior:

— `take(n,l)` computes the list containing the first `n` elements of list `l`.
— `from(n)` computes the (infinite) list containing all the consecutive natural numbers starting at `n`.

However, the function `from` is erroneous, because its body should be `n: from (S n)` instead of `n:from n`. The error becomes apparent when we evaluate the expression `take (S (S Z)) (from Z)`, which is expected to return the list containing the two first natural numbers, that is, `Z:S Z:[]`. However, the result returned is the wrong answer `Z:Z:[]`. An algorithmic debugger will internally build the EDT of Figure 6. In this tree, it is worthwhile to note that the occurrence of symbol ⊥ to represent a part of a structure that has not been evaluated during the computation. For instance, the node `from Z = Z:Z:`⊥ indicates that the function call `from Z` produced a list whose first and second elements are `Z`. Thus, when occurring on the right-hand side, the symbol ⊥ must be interpreted as an existential variable, and the question about the validity of this node

can be understood as "do you expect that `from Z` returns `Z:Z:⊥` for some value ⊥?" The answer is *no*, and thus the node is marked as invalid. Moreover, this node is buggy, because its only child `from Z = Z : ⊥` is valid, since we indeed expect `from Z` to produce a list with `Z` as first element. In this example, this is the only buggy node, which corresponds with function `from`, indicating that this function is defined with an incorrect rule.

On the other hand, ⊥ on the left-hand side has the implicit meaning of a universally quantified variable. For instance, the question about the validity of `take (S Z) (Z:⊥) = Z:[]` must be understood as "do you expect that for every list starting with `Z` the function call `take (S Z) (Z:⊥)` produces the result `Z:[]`?" This is true, and thus the node is valid.

Thus, the EDT reflects the philosophy of lazy functional computations by producing execution tree nodes where only those expressions that are demanded are evaluated to values.

Regarding the implementation, Nilsson and Sparud also considered the interesting question of *how* this tree could be obtained and proposed the two main alternatives discussed in Section 2.2.

The first alternative, a source to source transformation, was later explored in detail, giving rise to Buddha (Pope 1998, 2006), an algorithmic debugger for Haskell, and also algorithmic debuggers for multi-paradigm languages such as $\mathcal{T}oy$ and *Curry* (Caballero and Rodríguez-Artalejo 2002; Lux 2008). However, the lack of efficiency of these proposals when considering large realistic programs gave rise to new proposals based on the modification of the compiler, which solves this problem. This idea was developed in Nilsson (2001), where a very efficient algorithmic debugger for Haskell was presented. More recent works have followed these ideas, either in combination with other techniques such as program slicing (Silva and Chitil 2006) or improving the debugging process by considering only those program modules that may contain an error (Faddegon and Chitil 2015). Recent works (Chitil et al. 2016; Faddegon and Chitil 2016) suggest using functional traces as a simple and lightweight technique for obtaining the execution trees.

Although the EDT solves the issue of the nested calls, which are now replaced by their corresponding evaluations, the problem remains for partial function applications. Chitil and Davie (2008) show that the EDT can ask questions like `allOddC (allOddC id (Leaf 5)) (Leaf 7) True = False`?. In this question, `allOddC` is a function with three parameters. Thus, the inner call `(allOddC id (Leaf 5))` cannot be evaluated, since it is applied only to two parameters; that is, it is a partial function application. Moreover, this partial application includes another one, `id`, which corresponds to the identity function defined by the program rule `id x = x`. As the authors observe, the number of function symbol occurrences in a function call is unbounded, and similar questions arise frequently in higher-order functional programs, which can make algorithmic debugging unpractical. To solve this problem, Pope (2006) and Davie and Chitil (2006a) propose representing a functional value as a finite map from arguments to results. The new debugger asks questions of the shape `allOddC {False→False} (Leaf 5) True = False`?. The new notation simplifies the questions involving partial applications and defines a new execution tree, the *function dependency tree* (FDT). The formal definition of the FDT, combined with a proof of correctness and a comparative between FDTs and EDTs, can be found in Chitil and Davie (2008).

3.3 Imperative Programming

The first application of algorithmic debugging in the field of imperative programming is Kamkar et al. (1990). This proposal combines algorithmic debugging with the technique of program slicing (Harman and Hierons 2001) to find bugs in programs written in the imperative language Pascal (Jensen and Wirth 1974). The article also faces the important issue of how to deal with side-effects such as global variable modification, which did not exist in the pure declarative paradigm, but it is important in imperative and object-oriented programs. Consider, for instance, the Pascal function:

```
function f ( y : integer ) : integer ;
begin
  b := b+1;
  f := y*2 ;
end ;
```

A possible call to this function is `f(2)`, which returns `4`. A first idea could be to include the node `f(2) -> 4` in the debugging tree. However, this is insufficient, because this node does not contain information about the modification of `b`, a variable defined out of the scope of function `f`. In fact, this side-effect might be the source of the error and should be taken into account. The authors propose a program transformation that produces a side-effect-free program. Applying the same ideas to the function `f`, we obtain:

```
function f ( y : integer ; var b : integer ) : integer ;
begin
  b := b+1;
  f := y*2 ;
end ;
```

The reserved word `var` indicates that the parameter `b` is modified inside the function. Note that the transformation must also modify all the calls to `f` to consider the new extra parameter. A possible node representation might be `f(2) {b→2} -> 4 {b→3}`. The debugger "remembers" that the second argument corresponds to `b` and extracts this parameter creating a *context* that the oracle must take into account when checking the validity of the node.

A related article is Shahmehri and Fritzson (1990), which also considers a subset of Pascal and side-effects. This article also studies other particularities of imperative languages such as loops. The author proposes that loops should become part of the debugging tree. To achieve this, the program loops are converted into new functions. The nice outcome is that now loops can be buggy nodes; that is, a more fine-grained debugger is obtained.

However, this also poses a new problem. The new functions that replace the loops are recursive functions, and each iteration generates a new call and thus a new node in the tree. In the case of loops with many iterations, this gives rise to huge debugging trees with very lengthy branches. Another problem is that the questions associated with the new functions are difficult to answer, because they correspond to code not programmed by the user (i.e., code inserted by the program transformation). These problems were considered and different solutions were proposed in Insa et al. (2013a).

3.4 Object Oriented Programming

The same ideas of imperative languages are also applicable to object-oriented languages such as Java (Kouh and Yoo 2003; Caballero et al. 2006). The main difference is that the contexts tend to be more complicated, since the objects' states must be included. One problem introduced by objects is the fact that an object can contain itself as the value of an attribute. This produces an infinite recursive data structure that should be suitably displayed to the user. In this context, Kouh and Yoo (2003) proposed a hybrid approach that combines algorithmic and trace debugging. This idea has been further developed in later works (Hermanns and Kuchen 2013; Insa and Silva 2014).

The instance of algorithmic debugging applied to the object-oriented programming paradigm is summarized as follows:

(1) It considers method calls, and often loop executions, as possible incorrect computations.
(2) The result of the computation must consider the context, including global variables and object states.
(3) Each computation depends on the method calls and loop executions in its body.

4 OTHER APPLICATIONS OF ALGORITHMIC DEBUGGING

Beyond its usage as a program debugging method, the foundations of algorithmic debugging have been applied in other contexts. We outline here some applications of algorithmic debugging and briefly describe how this technique or its fundamentals have been used in other fields.

— *Tutoring systems.* Algorithmic debugging has been used in tutoring systems (Zinn 2014, 2013). Here, the debugger traverses the debugging tree produced by a *correct* program (thus, no buggy node exists) and, by asking questions to the students, it looks for the error in the intended interpretation, that is, a *buggy answer*, which corresponds to the function that the student does not understand.

— *Test case generation.* In the standard software development cycle, bugs are found during the testing phase; then debugging is applied until the implementation is considered correct; and, finally, all the knowledge obtained during this phase is discarded. To solve this problem, algorithmic debugging has also been used for test case generation. The first idea is that every node in the debugging tree is a potential test case, and the answers given by the oracle validate these test cases. A second idea is that, when a set of test cases fail, the debugging trees of all failing test cases can be generated and combined to (i) direct the search for a bug but also to (2) refine the test cases that fail, producing more specific test cases from the nodes of the generated trees.

  The first approaches (Fritzson et al. 1992; Kókai et al. 1997) combined debugging with the category-partition testing approach (Ostrand and Balcer 1988). In this approach, different *categories* are defined for the parameters (e.g., the categories for a list are its *size* and the *type* of the elements), and test cases are generated by combining one value from each category (one test case could take the *empty* and *positive* values from the *size* and *type* categories, respectively, while other test case could take *nonempty* and *positive*); the aim is to write test cases for all the (possibly restricted) combinations, although the huge number of combinations makes it unfeasible in general. By combining testing and algorithmic debugging, it is possible to prevent the debugger from asking the user questions that are already defined in the test cases, while it is also possible to extend the database of test cases to cover combinations that were not taken into account by the programmer.

  Modern approaches (Tamarit et al. 2016) use unit testing (Runeson 2006) in the same way, hence integrating debugging into the software development cycle. However, other testing approaches such as random testing (Hughes 2010) or mutation testing (Jia and Harman 2011) remain unused; integrating these novel testing frameworks would make algorithmic debugging more attractive for developers. On the one hand, random testing relies on properties, usually written in first-order logic, that some functions must satisfy. These properties can hence be also applied to the nodes of the debugging tree: If the property fails, then the node would be automatically marked as wrong, while if the property holds, then we can mark it as correct if this property only holds for those elements in the intended interpretation.[1] On the other hand, using mutation testing would provide a solid database of test cases that can be used in the same way as the unit tests described above.

— *Software Evolution Control.* A common drawback of most of the debugging techniques developed thus far is that they are not integrated into the software development cycle: for

[1] Although this discussion is beyond the scope of the article, it is easy to see the intuitive idea with an example: A property on a sorting algorithm that only requires the elements in the resulting list to be ordered cannot be used to mark nodes as correct; once we had the extra condition stating that the resulting list has the same elements as the original list the property is *complete* and can be used to mark nodes as correct.

example, although the spiral process model is often used and hence previous debugged versions of functions are available, they are not used for debugging. An interesting approach that takes this information into account consists of the recording of the debugging sessions and their use in future releases of the software to automatically check whether those pieces of code that worked well or that were already fixed still maintain the same desirable behavior. As outlined before, the recorded debugging sessions can be considered as repositories of test cases that can be used to automatically test the evolved software. This idea is very related to another approach that also records and profits the previous debugging sessions: relative debugging (Abramson et al. 1996), a technique that has been successfully applied to several environments (Rose et al. 2015; Dinh et al. 2014; Sosic and Abramson 1997). Relative debugging frameworks allow the user to establish a comparison between the execution of two programs. This is achieved by defining a correspondence between the two program states, and hence it is closely related to algorithmic debugging: former results can be used as the oracle for the questions in the current session. In fact, this approach is available in several debuggers, as shown in Section 7.

— *Debugging Attribute Grammars.* Instead of debugging programs, Sugavanam (2013) proposes algorithmic debugging for finding errors in the formalism known as *attribute grammars*. In this case, the debugging tree corresponds to a dependency graph of attribute instances, where an attribute instance is an occurrence of a grammar attribute in a particular position of the syntax tree defined by the grammar. To reduce the number of questions, the debugger considers a ranking of possible values of a property of the grammar attributes, productions, and non-terminal symbols. This information is used to direct the questions during the navigation phase.

— *Design validation and diagnosis of VLSI circuits.* Algorithmic debugging has been also applied to hardware systems. First, the work in Naganuma et al. (1994) shows how high-level circuit designs are translated into Prolog to apply algorithmic debugging. Once the circuit is built, algorithmic debugging has been used to detect faulty components (Kuchcinski et al. 1993). In this case, we can consider that a logical circuit is just a function where inner components are auxiliary functions. Hence, by analyzing how these auxiliary functions behave, we can detect faulty components and either fix them or, when it is not possible, detect weak spots and improve the production process.

5 ISSUES OF ALGORITHMIC DEBUGGING

This section provides a critical view of algorithmic debugging. In particular, we describe the main issues of algorithmic debugging along its lifetime. We also explain why some of them have prevented this technique to be used in the industry for a long time, and we also identify those areas where algorithmic debugging is still missing. The solutions that have been proposed to overcome these problems are presented in Section 6.

We have divided the main issues in three different groups: scalability problems, effectiveness and user experience, and completeness of algorithmic debuggers. In the following, we discuss each of them separately. Each issue has been identified with a label (e.g., *Issue-1*) so it later can be easily used to discuss the solutions proposed in the literature and the existent implementations.

5.1 The Scalability Problem

The scalability problem has been the main reason why many algorithmic debuggers were abandoned (this is the case, e.g., for Buddha (Pope 1998, 2006), Hat-Delta (Davie and Chitil 2006b), and DDJ (Insa and Silva 2010), among others). For this reason, scalability has also been a hot topic, and

a target of many researches, but we have only seen significant advances in recent years (they will be explained in detail in Section 6).

In algorithmic debugging, the debugging time is often approximated as:

$$T = DTG + \sum_{i=1}^{NQ} DQ_i, \tag{1}$$

where DTG represents the time needed to generate the debugging tree, NQ is the number of questions that the user must answer, and DQ_i represents the difficulty of question i, which is measured by the time needed to answer the question (Silva 2007).

Issue-1: Time Needed to Generate the Debugging Tree [DTG in (1)]. Generating the debugging tree always implies executing the program. The record of an execution contains a huge amount of information, and, unfortunately, storing all this information and producing the debugging tree is often a slow process. In some languages, this problem is implicit to the current technology and it cannot be solved with accurate implementations or efficient methods to store the execution trace. For instance, several algorithmic debuggers exist for Java (Schildt 2014), such as JavaDD (Girgis and Jayaraman 2006) and DDJ (Insa and Silva 2010), that use the *Java Platform Debugger Architecture* (JPDA) (Liang 1999). This platform can be used to monitor and control the execution of a program in the standard Java Virtual Machine. Nevertheless, JPDA suffers from time scalability problems and, thus, all debuggers that use it are also affected by the same problems.

Issue-2: Memory Needed to Store the Debugging Tree. The size of a debugging tree of a large computation may be even gigabytes, which means that it often does not fit in main memory. Therefore, a mechanism to store it efficiently in secondary memory is needed. This applies not only to the main memory but also to the graphic memory, which can also be overflowed when the debugger tries to draw a huge debugging tree. The debugger should implement some technique to dynamically control the amount of information sent to the graphic memory.

5.2 User Experience and Effectiveness

One of the main complaints of algorithmic debugger users is that the tools and interfaces provided to the user need to be improved. In particular, users complain about the lack of control they have over the debugging session, the number and difficulty of the questions, the need of integration with other tools, and the granularity level of the errors.

Issue-3: Amount and Difficulty of the Questions [NQ and DQ_i in (1)]. An algorithmic debugger can generate many questions before it finds the bug. Clearly, the number of questions directly depends on the navigation strategy used during the debugging tree traversal. If the strategy only uses structural information (i.e., it is limited to a search in the tree), then the number of questions is bound by the optimal strategy (D&Q), whose query complexity is $O(N \cdot log\ N)$. This limit is still too large for real computations.

Moreover, questions can be very difficult for the user. For instance, while debugging a compiler, the Mercury's algorithmic debugger (MacLarty 2005) asked a question of more than 1400 lines. Clearly, a modern algorithmic debugger should allow the user to skip questions and should provide a mechanism to choose the easiest questions first. It is also desirable that those questions that are semantically related are asked consecutively. This can help to speed up the debugging session, because the user already has the context of the question in mind (i.e., the knowledge acquired in the first question can be used to answer the second question when they refer to the same subcomputation or routine).

Issue-4: Rigidity and Loss of Control During the Navigation Phase. Navigation strategies always select the node that they consider will lead to finding the error quickly. But, of course, to make this decision, they do not handle the user's intuitions about what computations are more likely to be wrong. Therefore, a rigid strategy that does not provide the user with the ability to redirect the search for the bug can be frustrating (i.e., answering questions that the user may know *a priori* that will not lead to the bug). Basic features such as the *undo* option, or the possibility to store the debugging session to continue later, are often disregarded in academic prototypes but are important to improve the user's experience and thus the tool's success.

Even though many algorithmic debuggers allow the user to start the computation from a particular routine call (with provided inputs), in most of them, once the debugging session has started, the user cannot tell the debugger to concentrate on a particular suspicious call. A possible solution to this issue is to allow the user to freely explore the debugging tree. Not having the possibility to explore the debugging tree (unfortunately, most debuggers are just console-based) prevents the user from inspecting the computation.

Issue-5: Integration with Other Debugging Tools. Another issue is the fact that most algorithmic debuggers completely ignore the debugging tools of the development environment (e.g., (conditional) breakpoints, debug perspective, variable inspector panels, etc.). Moreover, they are often offered as a separate tool, and the user must switch from their traditional debugger to start algorithmic debuggers. We think that, on the contrary, algorithmic debuggers should aim to integrate with existing debuggers, for instance, allowing the user to switch from a trace debugger to the algorithmic debugger easily, while maintaining the same environment and appearance.

Issue-6: Bug Granularity. By definition, the algorithmic debugging technique reports a whole routine as buggy, and thus, the granularity of errors is a whole routine. Hence, the user is forced to manually continue the search for the bug inside the reported buggy routine. Note that, although algorithmic debugging is a dynamic analysis technique (i.e., it needs to run the program), the answer of the debugger is usually static (i.e., it points out a set of lines of source code as buggy without considering the concrete buggy execution). Even if the algorithmic debugger does not point to a complete function but just a part of it (e.g., a loop), it still may contain some lines of code that were not executed in the call identified as buggy. A modern algorithmic debugger should implement some dynamic analysis technique to further reduce the code reported as buggy.

5.3 Completeness

An important drawback of algorithmic debuggers is the existence of untreated features, in particular:

Issue-7: Termination. We are not aware of any algorithmic debugging technique that faces nontermination. A modern debugger should (at least) allow the user to stop an infinite computation and debug the part of the debugging tree generated so far.

Issue-8: Concurrency. Concurrency is missing in almost all algorithmic debuggers. The reason is the difficulty of storing concurrent computations (the standard debugging tree is not prepared for that), and the difficulty of asking the user about the behavior of a concurrent computation. To the best of our knowledge, EDD (Caballero et al. 2015) is the only algorithmic debugger able to debug concurrent programs.

Issue-9: I/O. In general, programs that access the file system, a database, or any other external source are not treated by algorithmic debuggers. The reason is that it is difficult for a debugger

(often impossible) to store the state of the external source when it was accessed,and, in general, this state is needed for the user to answer the question.

Example 5.1 (Accessing a database). Consider the function call:

```
getFromDataBasePeopleOlderThan(42)
```

Clearly, the user needs the database before and after this call to know if the result of the call and the final state of the database—it should remain unchanged—are correct.

Even in the case where the debugger is able to store/access the external source, it still has the problem of how to show it to the user. Although some solutions have been provided in the case of declarative languages (Lux 2008), the general question remains unsolved.

6 OPTIMIZATION AND USABILITY FEATURES OF ALGORITHMIC DEBUGGERS

In this section, we describe the current state of the art concerning the optimization and usability features of algorithmic debuggers. Hence, this section can be seen as a requirements specification for algorithmic debugger implementers. Moreover, because all the proposed features have been contextualized along the history of algorithmic debugging, this list is also a report about the solutions proposed so far to the issues described in Section 5. Besides the description of the modern features of algorithmic debuggers, we propose several ideas on how to overcome some untreated issues. The list of desirable features of an algorithmic debugger is the following.

6.1 Scalability [deals with *Issue-1* and *Issue-2*]

Nowadays, scalability is still one of the main problems of algorithmic debugging. In an industrial context, the debugging trees generated are huge (i.e., gigabytes), and, thus, they cannot be stored in main memory (*Issue-2*).

Unfortunately, many debuggers were designed to store the whole debugging tree in main memory and, hence, they crash at runtime (producing a "memory overflow" exception) when debugging real-size programs. There are solutions to this problem that have been implemented in several algorithmic debuggers. One solution implemented for declarative languages is to produce the debugging tree on demand (Nilsson 1998; MacLarty 2005). A second solution is to store the debugging tree in secondary memory (Davie and Chitil 2006b).

It is desirable to combine both approaches, thus keeping the speed gained when storing the tree in main memory while keeping the scalability provided by secondary memory. This is achieved by the approach in Insa and Silva (2011c), which keeps the whole debugging tree in secondary memory and caches a part of it in main memory. In this solution, the debugger only stores a cluster of nodes in main memory. This cluster is explored until a new cluster is needed (i.e., required by the navigation strategy used).

Another interesting approach has been integrated into the debugger B.i.O. (Braßel and Siegel 2008). B.i.O. avoids the space problem by not storing the debugging tree. This is done by reexecuting the program again and again to generate every single question during the debugging session. They also implement a mechanism to speed up the generation of questions. Because B.i.O. debugs Curry, which is a lazy language, the first time they execute the program, information on how much the subcomputations have been evaluated is recorded in a file. In the next executions, this file is used to execute the program again, but this time eagerly, to obtain the questions as they are required.

The solution of using a cache memory system has been also used for the graphical memory (*Issue-2*). The idea is to restrict the number of levels in the debugging tree shown in the graphical

interface. For this, the technique proposed in Insa and Silva (2011c) uses a *presentation cache* that can be parameterized to restrict the memory used in the GUI for the debugging tree.

To avoid the problem of waiting until the debugging tree has been completely generated before starting the debugging session (*Issue-1*), some debuggers implement the notion of *virtual debugging tree* (VDT) (Insa and Silva 2011c). Roughly, a VDT is an incomplete debugging tree (i.e., only some parts of the debugging tree have been executed, and thus some nodes are still empty). The idea is to allow the user to debug the program without having to wait until the debugging tree is complete. For this, the debugger uses two parallel threads. The first focuses on generating the debugging tree while the second on traversing it to find buggy nodes in the already-generated part.

6.2 Trusting Modules, Routines, and Arguments [deals with *Issue-1*, *Issue-2*, and *Issue-3*]

When programming, it is quite usual to reuse code (e.g., external routines). Hence, when we are debugging a program, this reused code should normally be trusted. The users should be able to trust external code but also parts of their own code. Ideally, the debugger should allow them to trust modules, routines, and arguments. And trusting should be able to be done:

> *Statically*. At compiling time, using anotations or flags. This allows the debugger to avoid the generation of nodes associated with trusted code, thus saving time (*Issue-1*) and space (*Issue-2*).
>
> *Dynamically*. During debugging time, when the oracle uses the answer "Trusted.". Then all the nodes corresponding to the same predicate/atom/function/procedure/method are automatically marked as valid (*Issue-3*).

6.3 Multiple Navigation Strategies [deals with *issue 3*]

The time required to find the bug is possibly the most important metric that can be used to measure the performance of any kind of debugger. In the case of an algorithmic debugger, this time strongly depends on the time spent by the oracle to check the validity of the nodes, which is related to the total number of questions, see Equation (1). Since the first articles on algorithmic debugging, different navigation strategies have been proposed to reduce the number and complexity of the questions. The most important strategies proposed are (chronologically):

—Single Stepping (Shapiro 1982a)
—Divide & Query (Shapiro 1982a)
—Top-Down Search (Lloyd 1987a)
—Top-Down Zooming (Maeji and Kanamori 1987)
—Hirunkitti's Divide & Query (Hirunkitti and Hogger 1993)
—Heaviest First (Binks 1995)
—Biased Weighting Divide & Query (MacLarty 2005)
—Subterm Dependency Tracking (MacLarty 2005)
—Hat Delta (Davie and Chitil 2006b)
—Less YES First (Silva 2006)
—Divide by YES & Query (Silva 2006)
—Dynamic Weighting Search (Silva 2006)
—Optimal Divide & Query (Insa and Silva 2011a)
—Speculative Divide & Query (Insa and Silva 2011b)
—Coverage-Based Search (Hermanns and Kuchen 2013)

Other strategies exist, but they are mostly variants of those in the list above. A deep explanation and analysis of the strategies with a theoretical and empirical comparative of them can be found in Silva (2007) and Silva (2011).

In general, all the strategies focus on three objectives:

(1) Reducing the number of questions (i.e., reducing *NQ* in Equation (1)). This is done by pruning the debugging tree (e.g., the strategy divide-and-query (Shapiro 1982a) tries to reduce the size of the tree by nearly a half after each answer).
(2) Reducing the time spent to answer the questions (i.e., reducing DQ_i in Equation (1)). This is done by avoiding difficult (i.e., complex or very verbose) questions or by consecutively asking questions that refer to related parts of the computation (i.e., consecutive questions are related to the same fragments of code). For instance, Top-Down Zooming (Maeji and Kanamori 1987) first considers nodes associated to the same recursive (sub)computation.
(3) Giving the user control over the search for the bug. For instance, the Subterm Dependency Tracking strategy (MacLarty 2005) gives the user the possibility of marking arguments, results, or even subexpressions, as suspicious. This information redirects the debugger to the nodes useful for tracking the suspicious terms.

Unfortunately, many old debuggers only implement one strategy, thus the programmer must follow a predefined and rigid order of questions. Fortunately, there has been a significant effort to solve this problem, and modern debuggers not only include many strategies but also implement hybrid and dynamic strategies (the strategy selected changes as the debugging tree does).

6.4 Accepted Answers [deals with *Issue-3*]

Algorithmic debugging strategies use the information provided by the oracle to prune the debugging tree following the schema of Figure 2. When the debugger selects a node *N* in the debugging tree, and it asks the question associated with this node, the oracle should be able to answer with at least the following options:

— "Yes" when the node is valid (i.e., correct). This answer removes the subtree rooted by *N* (see Proposition 2.3.2).
— "No" to indicate that the node is invalid (i.e., wrong). Then, the subtree with root *N* becomes the new debugging tree (see Proposition 2.3.1).
— "Inadmissible" (Pereira 1986) to indicate that some argument in the predicate/atom or function/procedure/method call associated with the question is wrong or suspicious (e.g., this computation should not take place because it does not satisfy the preconditions of the predicate/atom/function/procedure/method). For instance, if while debugging a program that computes the factorial of positive natural numbers we find a node of the form `fib(-45)` → `-37`, then we should mark this node as *inadmissible*. Note that answering "No" to this question makes the debugger continue the search inside the subtree of the node associated to this question. Contrarily, if we answer "Inadmissible" to the same question, then the search is redirected outside the subtree of this question (it searches for those nodes that can potentially influence the inadmissible argument) (Silva and Chitil 2006).
— "I don't know" to delay or skip a question (in general, because it is too difficult or because the user knows that it may not lead to the bug). These questions are avoided during the rest of the debugging session and only considered if there is no other choice, since they introduce non-completeness.
— "Trusted" to inform the debugger that this question can be considered correct (e.g., because it has been already proved correct or because it belongs to a module that has not been

recoded). The user should be able to trust modules, arguments, and functions/predicates/methods/ procedures. The nodes corresponding to computations associated with trusted modules, arguments, or function/predicate/method/procedure should also be considered correct (i.e., trusted) automatically.

6.5 Tracing Subexpressions [deals with *Issue-3* and *Issue-4*]

When the user knows what part of a question is wrong, answering "No" or "Inadmissible" becomes imprecise. In this case, they should be able to indicate that a subexpression is incorrect (instead of the whole question). This can help the debugger focus on the nodes related to this subexpression. As an example, Mercury's debugger (MacLarty 2005) uses information from the subexpressions to enhance the search for the bug.

The advantage of including this feature is twofold. First, the search space is reduced (*Issue-3*), because the debugger only explores those parts of the debugging tree that are actually related to the wrong subexpressions. Second, the debugging process becomes more understandable, because the user has more control over the search for the bug (*Issue-4*).

6.6 Debugging Tree Transformations [deals with *Issue-3*, and *Issue-6*]

Much of the work done in algorithmic debugging has focused on reducing the number of questions of a debugging session (*Issue-3*). In the early stages of algorithmic debugging, the efforts were concentrated on traversing the debugging tree efficiently (i.e., producing better traversing algorithms that visit fewer nodes before finding a buggy node). The result is a wide range of ideas and algorithms to traverse the debugging tree, often known as navigation strategies (see Section 6.3).

In 1990, Shahmehri and Fritzson (1990) noted that transforming the source code before generating the debugging tree can produce a better structure where it is easier to find a bug. It was not until 2006 that Davie and Chitil (2006b) proposed another way of reducing the number of questions: transforming the own debugging tree before the strategies traverse it. Later, David Insa and Josep Silva proposed a taxonomy (Insa and Silva 2015b) that classifies the transformations into three groups: (A) source code transformations, (B) execution transformations, and (C) debugging tree transformations. We describe in the following the main transformations proposed so far.

— *Loop expansion (Shahmehri and Fritzson 1990) (A)* relies on the observation that, in algorithmic debugging, deep debugging trees are easier to debug than wide debugging trees. Therefore, since iterative loops produce wide debugging trees and recursive functions produce deep debugging trees, recursion is better for algorithmic debugging than iteration. Hence, to produce debugging trees that can be debugged with fewer questions (*Issue-3*), a source code transformation from iterative loops to recursive functions was proposed. The term *loop expansion* was proposed in Insa et al. (2013b), where the authors proposed an algorithm to control when to expand iterations. For this, they proposed a more general transformation from iteration to recursion that also considers labels, exceptions, nested recursion, and so on (Insa and Silva 2015a).

  Loop expansion has another advantage: It reduces the granularity level of errors (*Issue-6*). The reason is that an iterative loop is represented with one node in the debugging tree. Contrarily, the recursive counterpart represents the loop with several nodes (often one for each iteration). Therefore, while the iterative version of a debugging tree only allows us to identify a whole loop as buggy, the recursive version allows us to identify buggy iterations.

— *Tree compression (Davie and Chitil 2006b) (C)* removes redundant nodes from the debugging tree preserving completeness. We illustrate this technique by showing a debugging tree and its compressed version. Let us consider the function definition of append shown in Figure 7.

```
(1) append [] y = y
(2) append (x:xs) y = x:append xs y
```

Fig. 7. The append function.

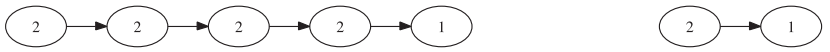

Fig. 8. Debugging tree of append `[1,2,3,4] [5,6]` (left) and its associated compressed tree (right) from Davie and Chitil (2006b).

Fig. 9. Debugging tree (left) and its completely balanced version (right). The grey nodes are *projection* nodes.

If we consider the call "append `[1,2,3,4] [5,6]`," then the debugging tree produced is the one in Figure 8 (left). In this example, append recursively calls the second rule 4 times, and finally uses the first rule (the base case). Hence, only two rules can be buggy, and we can compress the debugging tree producing the new tree shown in Figure 8 (right). In the compressed debugging tree, no matter what navigation strategy we use, the debugger asks a maximum of two questions.

Later, in Insa et al. (2013b), it was noted that tree compression should not be used indiscriminately. In some cases, compressing the tree increases the number of questions. They showed that, in general, the optimal debugging tree from the point of view of tree compression is one where some parts have been compressed and other parts remain unchanged (although they could be compressed). And, moreover, this optimal solution also depends on the navigation strategy used. As a consequence, they proposed an algorithm that can be used to dynamically transform the debugging tree during the debugging session, and this transformation is parameterized with the navigation strategy used.

Tree compression should be implemented by all the debuggers, and it should be applied after the debugging tree is produced, because it removes unnecessary questions before starting the debugging session at no cost.

— *Node simplification (Caballero et al. 2011) (C).* In line with the observations in Nilsson (2001) regarding the functional nature of algorithmic debugging, specific evaluation strategies (e.g., laziness) should be abstracted and not interfere with the debugging tree. For this reason, node simplification abstracts the execution details in proof trees by relating transformations applied on the same subexpression, so they can be put together later without affecting the completeness and correctness results. It deals with (*Issue-3*).

— *Tree balancing (Insa et al. 2013a) (C).* The objective of this transformation is twofold. On the one hand, it tries to transform the debugging tree into a binomial tree because in these trees navigation strategies can find a bug in logarithmic time. On the other hand, it tries to group nodes so a single answer can answer several questions at a time. Both objectives deal with *Issue-3*.

These objectives are achieved with the use of the so-called *projection* and *collapse* nodes. A projection node is a node that represents the combined execution of several sibling nodes, and it is inserted as their parent. For instance, in Figure 9 (left) we see a debugging tree whose root has four children. On the right, we see a balanced version, which contains two

projection nodes (in grey). When all the children of a projection node refer to the same rule of the program, then they all can be removed (if the projection node is buggy, it does not matter what children are buggy, because the buggy code is necessarily their associated rule). In this case, the projection node is called collapse node.

Note a fundamental difference between tree compression and tree balancing: While tree compression always removes nodes from the tree, tree balancing can increase the number of nodes with a projection, as in Figure 9. On the other hand, collapsed nodes can be actually seen as a tree compression.

6.7 Memoization [deals with *Issue-3* and *Issue-5*]

When the oracle answers a question, this question should not be asked again. This can be achieved by simply storing the oracle answers. Memoization can be intra-session or inter-session. The latter, however, must be done carefully, since different routines with the same name might produce unexpected results. One possibility is to store the user's information in the form of Unit test cases, a well-known formalism. As noticed in Tamarit et al. (2016), this approach not only decreases the number of questions that the user needs to consider during a debugging session (Issue-3); it also integrates the debugger with the testing framework (Issue-5). This is because a tester tool can now use the tests obtained from algorithmic debugging, and the debugger can employ all the available tests to avoid questions during the debugging sessions.

6.8 Debugging Tree Exploration [deals with *Issue-4*]

Some algorithmic debuggers limit the debugging session to an interview: the debugger generates questions and the user only answers these questions. This process is usually too rigid and sometimes frustrating. For instance, if the user has an intuition about the location of the bug in the debugging tree, then the debugger should allow the user to freely explore the tree. This way, the user has more control over the debugging tree exploration, and he/she can direct the search for the bug or just select a part of the tree on which the debugger should focus the questions.

Furthermore, a GUI can speed up a debugging session, because it allows the user to freely explore the debugging tree and mark nodes regardless of (or in parallel with) the running strategy. Graphical features such as collapsing subcomputations of the debugging tree can be very useful.

6.9 Undo Capabilities [deals with *Issue-4*]

Users can make mistakes when they program, and they can also make mistakes when they debug. Thus, they should be allowed to rectify. In this respect, one desirable functionality is allowing users to undo their actions and, in particular, their answers. It is quite surprising that most algorithmic debuggers do not provide this feature, and, thus, users are forced to restart the whole debugging session when they answer a question incorrectly.

It is worth mentioning that some debuggers (e.g. Freja (Nilsson 2001)) allows the user to answer *maybe yes* and *maybe no* (besides *yes* and *no*). When this happens, the debugger records that no definitive answer has been given. Hence, later if the bug cannot be found with the answers provided, the debugger can return and ask those questions answered with *maybe* again.

6.10 Communication with an IDE [deals with *Issue-5* and *Issue-6*]

Modern algorithmic debuggers have been integrated into an Integrated Development Environment (IDE). Notable cases are JHyde (Hermanns and Kuchen 2013) and HDJ (Insa and Silva 2014). The IDE allows them to take advantage of the panels, tools, perspectives, and other features such as variable watching, code coloring, and so on. Moreover, interfacing the IDE allows these

Table 1. Relation Between Issues and Solutions

Feature \ Issue	1	2	3	4	5	6	7	8	9
Navigation strategy			✓						
Answers			✓						
Trac. subexp.			✓	✓					
DT transf.			✓			✓			
Memoization			✓		✓				
DT exploration				✓					
Undo				✓					
Trusting	✓	✓	✓						
Scalability	✓	✓							
Levels of errors			✗			✓			
Program slicing						✓			
IDE					✓	✓			

debuggers to communicate with other debuggers, producing hybrid approaches that increase usability (*Issue-5*). Hybrid approaches have the advantage that, once the buggy function has been identified, the debugger can continue the search inside this function, and, thus, the granularity of errors is potentially reduced to lines or expressions instead of functions (*Issue-6*).

6.11 Different Levels of Errors [deals with *Issue-6*]

The pieces of code pointed out as buggy by algorithmic debuggers can be too big. It is useful to know that a particular Java method is wrong but the programmer still needs to look for the concrete bug inside the method. Some debuggers, like (Caballero et al. 2015), give the programmer the possibility of going deeper, assuming that inner components of the piece of code such as loops or branch statements have their own intended behavior.

6.12 Program Slicing [deals with *Issue-6*]

Another way of increasing the granularity level of the error is by using *program slicing* (Tip 1995). This method allows the debugger to extract the exact portion of code that actually participated in the buggy execution (thus, discarding the code that cannot be the cause of the bug) (Silva and Chitil 2006).

6.13 Conclusions

Summarizing, we present in Table 1 the relation between the issues discussed in Section 5 and the features presented in this section. The mark ✓ indicates that the feature solves the problem, while ✗ indicates that the feature worsens it.

Clearly, reducing the number and difficulty of questions (*Issue-3*) has received more attention, especially in the early implementations and techniques developed for algorithmic debugging. In contrast, unfortunately, completeness issues still remain unsolved. This global view of the state of the art reveals that there is much room for research, and it provides directions and trends for that research. Finally, it is interesting to note that solving some problems (improving the granularity when locating bugs) worsens others (the complexity and number of questions), so we must consider the choices of our implementations carefully and probably allow the users to choose those that they want.

7 CURRENT STATE-OF-THE-ART ALGORITHMIC DEBUGGERS

In this section, we compare all mature algorithmic debuggers. This comparison, which includes both chronological information and their functional features, provides a roadmap of the evolution of algorithmic debugging. Hence, it allows us to study how and when algorithmic debugging has been expanded to other paradigms and languages and which of the techniques presented in the previous section have actually been implemented, (and, if so, when they were integrated into a real debugger). In the first part of our study, we selected a collection of algorithmic debuggers. After checking the latest implementations and discussing with the implementors, we found 14 algorithmic debuggers. Our goal is not to compare techniques for algorithmic debugging but to compare mature and usable implementations. Hence, in our study, we have evaluated the last implementation of the debuggers, not their last thesis/article/report description. The debugging tools included in this study are the following (sorted by paradigm):

Logic paradigm

—**NUDE** (NU-Prolog Debugging Environment) (Naish et al. 1989), first released in 1987, implements a set of debugging tools for NU-Prolog programs. It introduced an integration with static analysis, testing, and editing. Moreover, unreleased experimental versions had other features such as assertions and a flexible oracle.

Functional paradigm

—**Freja** (Chitil et al. 2001), first released in 1997 (although some prototypes were available in 1992), is a debugger for a subset of Haskell. Freja introduced piecemeal trace generation, which allowed for saving memory (which was quite slow at the time), a transformation for abstracting lazy evaluation, and support for list comprehensions, higher-order functions, and closures. It also introduced the answers Maybe Yes and Maybe No.
—**Hat** (the Haskell tracer) (Chitil et al. 2001), first released in 2000, consists of a number of tools that include Hat-Detect, an algorithmic debugger for Haskell. The algorithmic debugger Hat-Delta evolved from Hat-Detect, and it includes features such as tree compression and improved strategies to explore the ET.
—**Buddha** (Pope 2006), first released in 2003, was designed to debug Haskell 98 programs. It introduced support for higher-order functions in extensional style and debugging of I/O computations.
—**EDD** (Erlang Declarative Debugger) (Caballero et al. 2015), first released in 2013, is an algorithmic debugger that supports both sequential and concurrent Erlang programs. It introduced zoom-debugging, a way to inspect the code of a function previously detected as buggy, and a technique for combining unit tests and algorithmic debugging.

Functional-Logic paradigm

—**Münster Curry Debugger**, first released in 2002, is an algorithmic debugger integrated into the Münster Curry Compiler (Lux 2006).
—**DDT** (Caballero 2005), first released in 2002, belongs to the standard distribution of the multiparadigm language TOY (López-Fraguas and Sánchez-Hernández 1999).
—**Mercury's debugger**, first released in 2005, integrates both a procedural debugger and an algorithmic debugger (MacLarty 2005) for Mercury (Henderson et al. 2014).
—**B.i.O.** (Believe in Oracles) (Braßel and Siegel 2008), first released in 2008, is a debugger integrated into the Curry compiler KICS and can work as an algorithmic debugger. This algorithmic debugger has the peculiarity that it does not need to store the debugging tree,

but information on the execution, which allows B.i.O. to generate the questions on-the-fly and save memory, as explained in Section 6.1.

Declarative paradigm

—**DES** (Caballero et al. 2012), first released in 2007, is an algorithmic debugger for debugging SQL (Beaulieu 2005) views. It introduced slicing oriented to databases to minimize the number of tuples displayed to the user.
—**MDD** (Maude Declarative Debugger) (Riesco et al. 2012), first released in 2008, is an algorithmic debugger for Maude (Clavel et al. 2007) with support for functional and system modules. It introduced balancing techniques (introducing new nodes) for enhancing the divide-and-query strategy and debugging and trusting of sorts and normal forms (used when debugging missing answers).

Imperative paradigm

—**DDJ** (Insa and Silva 2010), first released in 2009, is a declarative debugger that supports all Java features. Its implementation, including an external database for storing the debugging tree, and its support for depicting objects makes DDJ the first complete algorithmic debugger for an imperative language.
—**JHyde** (Hermanns and Kuchen 2013), first released in 2011, is a hybrid debugger for Java that integrates algorithmic debugging and omniscient debugging. It was the first hybrid debugger including algorithmic debugging.
—**HDJ** (Hybrid Debugger for Java) (Insa and Silva 2014), first released in 2013, is an Eclipse plugin that combines trace debugging, algorithmic debugging, and omniscient debugging for debugging Java programs. Since HDJ integrates most of the features from all other algorithmic debuggers, it is integrated into a real IDE and combines other debugging techniques. It also the first to make algorithmic debugging an alternative for traditional debugging techniques.

Tables 2 and 3 summarize the features provided by the studied algorithmic debuggers. Each column corresponds to one algorithmic debugger. The rows have the following meaning:

—**Target Language**: It is the language targeted by the debugger.
—**Imp. Language**: It is the language used in the debugger implementation.
—**Strategies**: It is a list of the navigation strategies implemented by the debugger: Top Down (TD), Divide & Query (DQ), Hat-Delta's Heuristics (HD), Biased Weighting Divide & Query (BW), Single Stepping (SS), Coverage-based navigation (CB), and Subterm Dependency Tracking (SD).
—**DataBase/Memoization**: It indicates whether a database is used to memoize answers for future debugging sessions (inter-session memoization). It also indicates whether the answers are remembered during the same session (intra-session memoization).
—**Debugging Tree**: It indicates whether the debugging tree is saved as a file, or, on the contrary, it is stored in main memory (during the debugging session). In addition, it indicates whether the debugger produces the DT on demand.
—**Accepted Answers**: Yes (YES), No (NO), Don't Know (DK), Inadmissible (IN), Maybe Yes (MY), Maybe Not (MN), and Trusted (TR).
—**Tracing Subexpressions**: It indicates whether the user can mark a subexpression as incorrect.
—**Granularity**: Is it possible to find different levels of errors (e.g., inside buggy functions)?
—**DT Exploration**: It indicates whether the DT can be explored freely.

Table 2. Comparison of Algorithmic Debuggers I

Feature \ Debugger	NUDE	Freja	Hat-Delta	Buddha	EDD	Münster Curry Debugger	DDT
Target language	NU-Prolog	Haskell subset subset	Haskell 98	Haskell	Erlang	Curry	Toy
Imp. language	NU-Prolog	Haskell	Haskell	Haskell	Erlang	Haskell (front-end) Curry (back-end)	Prolog
Strategies	TD	TD	TD HD	TD	TD DQ	TD DQ	TD DQ
DataBase / Memoization	YES/YES	NO/NO	NO/YES	NO/YES	YES/YES	NO/YES	YES/YES
Debugging tree	Main memory on demand	Main memory	File	Main memory on demand	Main memory	Main memory	Main memory
Accepted answers	YES NO TR	YES NO MY MN	YES NO DK	YES NO DK IN TR	YES NO DK IN TR	YES NO	YES NO TR DK
Tracing subexpressions?	NO	NO	NO	NO	YES	NO	NO
Granularity	NO	NO	YES	NO	YES	NO	YES
DT exploration	NO	YES	YES	YES	YES	YES	YES
Transformations	-	LZY LST	TC	-	-	-	-
Early start	NO	NO	NO	NO	NO	NO	NO
Undo	YES	YES	YES	NO	YES	YES	YES
Trusting	Fun	Mod Fun	Mod	Mod Fun	Fun	Mod Fun	Fun
External oracle	NO	NO	NO	NO	YES	NO	YES
GUI	NO	NO	YES	NO	YES	NO	YES
Version	NU-Prolog 1.6.9 (1995)	March 2000	Hat 2.9 (July 2016)	Buddha 1.2.1 (01.12.2006)	November 2015	MCC 0.9.11 (June 2007)	DDT 2.0 (2004)

Table 3. Comparison of Algorithmic Debuggers II

Feature \ Debugger	Mercury Debugger	B.i.O.	DES	MDD	DDJ	JHyde	HDJ
Target language	Mercury	Curry	Datalog SQL	Maude	Java	Java	Java
Imp. language	Mercury / C	C Curry Haskell Prolog	Prolog	Maude Java	Java	Java	Java
Strategies	TD DQ BW SD	TD DQ SS	TD DQ	TD DQ SS	HD HD	TD DQ CB	HD HD
DataBase / Memoization	YES/YES	NO/NO	NO/YES	NO/YES	NO/YES	NO/NO	NO/YES
Debugging tree	Main memory on demand	Only oracle stored	External database	Main memory on demand	Main memory on demand External Database	Main memory	Main memory on demand External Database
Accepted answers	YES NO DK IN TR	YES NO TR DK	YES NO TR DK	YES NO DK IN TR	YES NO TR DK	YES NO TR	YES NO TR DK
Tracing subexpressions?	YES	NO	NO	NO	NO	NO	NO
Granularity	NO	YES	YES	NO	YES	YES	YES
DT exploration	NO	NO	NO	YES	YES	YES	YES
Transformations	-	-	-	TB	TC TB LE	-	TC TB
Early start	YES	YES	NO	YES	YES	NO	YES
Undo	YES	YES	YES	YES	YES	NO	YES
Trusting	Mod Fun	Fun	Mod	Mod Fun	Mod	Mod Fun	Mod
External oracle	NO	NO	NO	YES	NO	NO	NO
GUI	NO	NO	YES	YES	YES	YES	YES
Version	Mercury 14.01.1 (2014)	Kics 0.81893 (May 2008)	DES 4.1 (April 2016)	MDD 2.1 (February 2013)	DDJ 2.6 (April 2012)	JHyde 1.0 (2011)	HDJ 1.1 (October 2014)

—**Transformations**: It indicates whether the debugger implements transformation techniques to simplify the debugging tree. These techniques include Tree Compression (TC), Tree Balancing (TB), Loop Expansion (LE), for lazy evaluation (LZY), and for list comprehensions (LST).
—**Early start**: Is it possible to start the debugging process while the tree is being computed?
—**Undo**: Is it possible to undo an answer?
—**Trusting**: Is it possible to trust modules (Mod), functions (Fun), and/or arguments (Arg)?
—**External oracle**: Is it possible to use an external oracle to answer the questions?
—**GUI**: Does the debugger have a graphical user interface?
—**Version**: The version of the debugger that has been evaluated.

Summarizing, most of the debuggers have been developed for declarative languages that are not widely used beyond academia. Only since the early 2010s have debuggers for more used languages (Java and Erlang) been developed. Some of these debuggers (JHyde and HDJ) have an Eclipse plugin, which greatly eases its usage. Regarding memory, most of the debuggers store the debugging tree in main memory. Storing the tree in an external database (which is only done by three debuggers) improves the scalability, since it allows for storing huge trees. Hence, only these three debuggers can deal with real (industrial) situations without running out of memory. However, if the traversal starts after the complete tree has been computed, then it might require much time. For this reason, a complementary feature that saves time and space is the capability of starting the debugging process with an incomplete debugging tree; despite its importance, only four debuggers implement this feature. Similarly, only three debuggers can use an external oracle, and only EDD allows for test cases as oracles (the two others just allow the possibility of replacing the user answer by the result obtained using a previous, trusted version of the same program). Since testing is an established practice in software engineering, it would be worthwhile to expand this practice to other tools. On the other hand, we also note that (probably due to their temporal evolution) the tree transformations are missing in most of the debuggers. As a general requirement, all debuggers should implement the tree transformations, because they produce simpler and smaller trees at almost no cost. We also find that some features have improved from the early days of algorithm debugging: In general, debuggers provide a graphical user interface (eight tools), have different levels of granularity (eight tools), and allow for freely navigating the tree (seven tools). We expect that future tools will include these features. On the bright side, most of the tools have some kind of trusting mechanism (12 tools), implement at least two navigation strategies (13 tools), have some kind of memoization (13 tools), and have implemented an undo command (10 tools).

8 CONCLUSIONS

At the moment of writing this article, 35 years have passed since E. Y. Shapiro introduced algorithmic debugging in the context of logic programming. In these years, Shapiro's ideas have evolved into a general and fruitful debugging technique. As we have seen in Sections 1–3, functional, multiparadigm, imperative, object oriented, and even database query languages have developed their own versions of the general schema. All of them present their specific particularities, but at the same time they share the same common debugging principles. Moreover, we showed in Section 4 that similar ideas have been applied in other contexts, including tutoring systems, software evolution control, design validation of VLSI circuits, attribute grammars, and test-case generation. Thus, it can be claimed that algorithmic debugging has been, and still is, a very successful programming technique.

However, a critical analysis must also point out that, in spite of the many developed tools, algorithmic debugging has not reached the mainstream. Today, like 35 years ago, programmers spend

most of their working time struggling with trace-based debuggers, while almost none of them has even heard about algorithmic debugging. For this reason, we have devoted Sections 5–7 to discuss the main issues that affect algorithmic debugging and possible solutions. We have also reviewed the systems developed during these years and their main characteristics.

Regarding the problems presented in Section 5, the scalability problems suffered by the first algorithmic debuggers have been solved. Now, there exist sophisticated mechanisms to store the debugging tree in disk and load clusters of it on demand. The user experience has also been improved, although there are still some open issues. Even though the new navigation strategies greatly reduce the number and difficulty of the questions asked to the oracle, most of the available debuggers correspond to prototypes developed for academic research, with several limitations in the language features supported, lack of graphical support, and a very restricted set of user options. We think that the future of this technique depends on the development of mature and scalable algorithmic debuggers that focus on the completeness with respect to the supported language and on the user experience, including features such as the free manual navigation of the debugging tree, possibility of trusting functions and/or modules, and loading and saving debugging sessions. Finally, the completeness problems remain open; we have presented some ways to solve them if algorithmic debugging is combined with other debugging techniques, so we hope future algorithmic debuggers will solve these problems.

Algorithmic debugging has been historically applied to declarative languages for a number of reasons: their philosophy follows that of algorithmic debugging (i.e., abstracting evaluation details), they are pure and avoid low-level features (such as pointers), and they are good for prototyping and testing new formalisms. However, the problems that posed the absence of these features in imperative languages have been progressively solved and nowadays the most mature algorithmic debuggers are those for Java. In fact, once these problems could be solved, commercial languages such as Java or C became the most appropriate languages for implementation, since they have several libraries that ease tasks such as program analysis and development of graphical interfaces. Moreover, they have interactive development environments that include other debugging techniques (e.g. trace debugging), hence providing a good basis for integration.

Debugging is a formidable task, and future debugging systems will look for the cooperation of different tools. Moreover, new technologies such as *Big Data* (Chen et al. 2014) or *streaming* (Silva et al. 2013) pose new challenges that are not solved by traditional debuggers and require more powerful debugging tools. For example, in the case of algorithmic debuggers for Big Data, an important difficulty is that queries can involve very large datasets. In these cases, the debugger should allow the user to point out a particular erroneous row or document in the query output. This information could be used by the debugger to focus on the (hopefully small) subset responsible for the particular error using lineage techniques, in the line of the SQL debugger of Caballero et al. (2015). For streaming technologies, an algorithmic debugger might first distinguish whether the error is due to the batch computations or to the sequential treatment, and then focus of the functions responsible for the error. Thus, we suggest that, instead of designing algorithmic debuggers as alternatives to traditional debuggers, future tools should aim for the integration with already-existing trace debuggers and, in general, with coordinated debugging and testing frameworks. In fact, this integration would solve some open issues above. A combination of breakpoints and algorithmic debugging would solve (at least partially) the issue of termination: The debugging tree would be built for the functions called up to the breakpoint and, if no buggy node is found, then the process would continue until the next breakpoint. Regarding concurrency, this is a difficult issue that can benefit in many ways: assertions might stop the execution as soon as the program starts to fail, while static analysis techniques might reduce the number of questions posed to the user; finally, once the problem has been reduced, a graphical interface would greatly

help to understand how different processes/threads interact, hence allowing the user to direct the debugging process in the most appropriate way. Last, we explained that algorithmic debugging did not deal with I/O problems due to the impossibility in general of storing the complete state when dealing with external files, such as databases; perhaps combining algorithmic debugging with slicing (Tip 1995), which narrows down the parts of the program/memory related to some variables of interest (in our case those producing the wrong result) might greatly reduce the size of the data, hence allowing us to ask about these external files.

In this way, we think that algorithmic debuggers will reach a broader public and that they will play an important role in future debugging systems.

ACKNOWLEDGMENTS

The authors greatly thank Bernd Braÿel, Olaf Chitil, Sebastian Fisher, Herbert Küchen, David Insa, Wolfgang Lux, Ian MacCarty, Lee Naish, Henrik Nilsson, Bernie Pope, and Salvador Tamarit for providing detailed and useful information about their debuggers. We also thank David Insa for careful reading of our manuscript and insightful comments, Ashley J Naveso Cranford for a detailed English revision, and to the anonymous referees for their constructive criticism and suggestions.

REFERENCES

D. Abramson, I. Foster, J. Michalakes, and R. Sosič. 1996. Relative debugging: A new methodology for debugging scientific applications. *Commun. ACM* 39, 11 (Nov. 1996), 69–77. DOI: http://dx.doi.org/10.1145/240455.240475

K. R. Apt, H. A. Blair, and A. Walker. 1988. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, 89–148.

T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava. 1993. Explaining program execution in deductive systems. In *Deductive and Object-Oriented Databases*. Lecture Notes in Computer Science, Vol. 760. Springer-Verlag, Berlin, 101–119.

E. Av-Ron. 1984. *Top-Down Diagnosis of Prolog Programs*. Ph.D. Dissertation. Weizmann Institute.

A. Beaulieu. 2005. *Learning SQL*. O'Reilly, Farnham, UK.

D. Binks. 1995. *Declarative Debugging in Gödel*. Ph.D. Dissertation. University of Bristol.

B. Braßel and H. Siegel. 2008. *Debugging Lazy Functional Programs by Asking the Oracle*. Springer-Verlag, Berlin, 183–200. DOI: http://dx.doi.org/10.1007/978-3-540-85373-2_11

R. Caballero. 2005. A declarative debugger of incorrect answers for constraint functional-logic programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*. ACM Press, New York, NY, 8–13. DOI: http://dx.doi.org/10.1145/1085099.1085102

R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. 2008. A theoretical framework for the declarative debugging of datalog programs. In *Proceedings of the International Workshop on Semantics in Data and Knowledge Bases (SDKB'08)*. Lecture Notes in Computer Science, Vol. 4925. Springer-Verlag, Berlin, 143–159.

R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. 2012. Declarative debugging of wrong and missing answers for SQL views. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS'12)*. Lecture Notes in Computer Science, Tom Schrijversand Peter Thiemann (Eds.), Vol. 7294. Springer-Verlag, Berlin, 73–87. DOI: http://dx.doi.org/10.1007/978-3-642-29822-6_9

R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. 2015. Debugging of wrong and missing answers for datalog programs with constraint handling rules. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*. ACM, New York, NY, 55–66. DOI: http://dx.doi.org/10.1145/2790449.2790522

R. Caballero, C. Hermmans, and H. Kuchen. 2006. Algorithmic debugging of java programs. In *Proceedings of the 2006 Workshop on Functional Logic Programming (WFLP'06)*. Electronic Notes in Theoretical Computer Science. Elsevier, North-Holland, 63–76.

R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. 2015. A zoom-declarative debugger for sequential erlang programs. *Sci. Comput. Program.* 110 (2015), 104–118. DOI: http://dx.doi.org/10.1016/j.scico.2015.06.011

R. Caballero, A. Riesco, A. Verdejo, and N. Martí-Oliet. 2011. Simplifying questions in Maude declarative debugger by transforming proof trees. In *21st International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR'11)*. Lecture Notes in Computer Science, Germán Vidal (Ed.), Vol. 7225. Springer-Verlag, Berlin, 73–89.

R. Caballero and M. Rodríguez-Artalejo. 2002. A declarative debugging system for lazy functional logic programs. *Electr. Not. Theoret. Comput. Sci.* 64 (2002), 113–175.

S. Ceri, G. Gottlob, and L. Tanca. 1989. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (Mar. 1989), 146–166.

M. Chen, S. Mao, and Y. Liu. 2014. Big data: A survey. *Mob. Netw. Appl.* 19, 2 (Apr. 2014), 171–209. DOI: http://dx.doi.org/10.1007/s11036-013-0489-0

O. Chitil and T. Davie. 2008. Comprehending finite maps for algorithmic debugging of higher-order functional programs. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'08)*. ACM, New York, NY, 205–216. DOI: http://dx.doi.org/10.1145/1389449.1389475

O. Chitil, M. Faddegon, and C. Runciman. 2016. A lightweight hat: Simple type-preserving instrumentation for self-tracing lazy functional programs. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages (IFL'16)*. ACM, New York, NY, Article 10, 14 pages. DOI: http://dx.doi.org/10.1145/3064899.3064904

O. Chitil, C. Runciman, and M. Wallace. 2001. *Freja, Hat and Hood—A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*. Springer, Berlin, 176–193.

O. Chitil, C. Runciman, and Malcolm Wallace. 2003. *Transforming Haskell for Tracing*. Springer-Verlag, Berlin, 165–181. DOI: http://dx.doi.org/10.1007/3-540-44854-3_11

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. 2007. *All About Maude: A High-Performance Logical Framework*. Lecture Notes in Computer Science, Vol. 4350. Springer, Berlin.

T. Davie and O. Chitil. 2006a. Display of functional values for debugging. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06)*. 326–337.

T. Davie and O. Chitil. 2006b. Hat-delta: One right does make a wrong. In *Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06)*. Intellect, Bristol, UK, 1–9.

M. Ngoc Dinh, D. Abramson, and C. Jin. 2014. Scalable relative debugging. *IEEE Trans. Parallel Distrib. Syst.* 25, 3 (2014), 740–749. DOI: http://dx.doi.org/10.1109/TPDS.2013.86

M. Faddegon and O. Chitil. 2015. Algorithmic debugging of real-world haskell programs: Deriving dependencies from the cost centre stack. *SIGPLAN Not.* 50, 6 (Jun. 2015), 33–42.

M. Faddegon and O. Chitil. 2016. Lightweight computation tree tracing for lazy functional languages. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM Press, New York, NY, 114–128.

G. Ferrand. 1987. Error diagnosis in logic programming an adaptation of E.Y. shapiro's method. *J. Logic Program.* 4, 3 (1987), 177–198.

P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. 1992. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.* 1, 4 (1992), 303–322.

M. P. J. Fromherz. 1993. Towards declarative debugging of concurrent constraint programs. In *Proceeedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG'93)*. Springer-Verlag, Berlin, 88–100.

H. Girgis and B. Jayaraman. March 2006. *JavaDD: A Declarative Debugger for Java*. Technical Report 2006-07. University at Buffalo.

M. Harman and R. M. Hierons. 2001. An overview of program slicing. *Softw. Focus* 2, 3 (2001), 85–92.

F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, M. Brown, and P. Wang. 2014. *The Mercury Language Reference Manual (Version 14.01.1)*. The University of Melbourne.

C. Hermanns and H. Kuchen. 2013. *Hybrid Debugging of Java Programs*. Springer-Verlag, Berlin, 91–107. DOI: http://dx.doi.org/10.1007/978-3-642-36177-7_6

V. Hirunkitti and C. J. Hogger. 1993. A generalised query minimisation for program debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*. Lecture Notes in Computer Science, Vol. 749. Springer-Verlag, Berlin, 153–170.

J. Hughes. 2010. Software testing with quickcheck. In *Proceedings of the 3rd Summer School Conference on Central European Functional Programming School, CEFP 2009*, Z. Horváth, R. Plasmeijer, and V. Zsók (Eds.), Vol. 6299. Springer-Verlag, Berlin Heidelberg, 183–223.

M. H. Huntbach. 1987. Algorithmic PARLOG debugging. In *Proceedings of the 1987 Symposium on Logic Programming*. IEEE Comput. Soc. Press, New York, NY, 288–297.

G. Hutton. 2016. *Programming in Haskell*. Cambridge University Press, Cambridge, UK.

D. Insa and J. Silva. 2010. An algorithmic debugger for java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, Michele Lanzaand Andrian Marcus (Eds.). IEEE Computer Society, New York, NY, 1–6.

D. Insa and J. Silva. 2011a. Optimal divide and query. In *Proceedings of the 15th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (EPIA'11)*. Lecture Notes in Computer Science, Vol. 7026. Springer-Verlag, Berlin, 224–238. DOI: http://dx.doi.org/10.1007/978-3-642-24769-9_17

D. Insa and J. Silva. 2011b. An optimal strategy for algorithmic debugging. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. IEEE, New York, NY, 203–212. DOI: http://dx.doi.org/10.1109/ASE.2011.6100055

D. Insa and J. Silva. 2011c. *Scaling Up Algorithmic Debugging with Virtual Execution Trees*. Springer-Verlag, Berlin, 149–163. DOI: http://dx.doi.org/10.1007/978-3-642-20551-4_10

D. Insa and J. Silva. 2014. A new hybrid debugging architecture for eclipse. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*. Lecture Notes in Computer Science, Gopal Guptaand Ricardo Peña (Eds.), Vol. 8901. Springer-Verlag, Berlin, 183–201.

D. Insa and J. Silva. 2015a. Automatic transformation of iterative loops into recursive methods. *Information & Software Technology* 58 (2015), 95–109. DOI: http://dx.doi.org/10.1016/j.infsof.2014.10.001

D. Insa and J. Silva. 2015b. A generalized model for algorithmic debugging. In *Proceedings of the Logic-Based Program Synthesis and Transformation: 25th International Symposium (LOPSTR'15)*. Lectures Notes in Computer Science, Moreno Falaschi (Ed.). Springer-Verlag, Berlin, 261–276.

D. Insa, J. Silva, and A. Riesco. 2013a. Speeding up algorithmic debugging using balanced execution trees. In *Proceedings of the 7th International Conference on Tests and Proofs (TAP'13)*. Lecture Notes in Computer Science, Vol. 7942. Springer-Verlag, Berlin, 133–151.

D. Insa, J. Silva, and C. Tomás. 2013b. *Enhancing Declarative Debugging with Loop Expansion and Tree Compression*. Lecture Notes in Computer Science, Vol. 7844. Springer-Verlag, Berlin, 71–88. DOI: http://dx.doi.org/10.1007/978-3-642-38197-3_6

K. Jensen and N. Wirth. 1974. *PASCAL User Manual and Report*. Springer-Verlag, Berlin.

Y. Jia and M. Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649–678. DOI: http://dx.doi.org/10.1109/TSE.2010.62

M. Kamkar, N. Shahmehri, and P. Fritzson. 1990. Bug localization by algorithmic debugging and program slicing. In *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming (PLILP'90)* Lecture Notes in Computer Science, Vol. 456. Springer-Verlag, Berlin, 60–74.

S. Köhler, B. Ludäscher, and Y. Smaragdakis. 2012. *Declarative Datalog Debugging for Mere Mortals*. Springer-Verlag, Berlin, 111–122.

G. Kókai, L. Harmath, and T. Gyimóthy. 1997. Algorithmic debugging and testing of prolog programs. In *The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments (ICLP'97)*. Leuven, Belgium, 14–21.

H. Kouh and W. Yoo. 2003. The efficient debugging system for locating logical errors in java programs. In *Proceedings of the 2003 International Conference on Computational Science and Its Applications: Part I (ICCSA'03)*. Springer-Verlag, Berlin, 684–693.

R. Kowalski. 2014. Logic programming. In *Handbook of the History of Logic, Volume 9—Computational Logic*, Dov M. Gabbay, Jörg H. Siekmann, and John Woods (Eds.). Elsevier, Amsterdam, 215–254. DOI: http://dx.doi.org/10.1016/B978-0-444-51624-4.50005-8

R. Kowalski and D. Kuehner. 1971. Linear resolution with selection function. *Artif. Intell.* 2, 3–4 (Dec. 1971), 227–260. DOI: http://dx.doi.org/10.1016/0004-3702(71)90012-9

K. Kuchcinski, W. Drabent, and J. Maluszynski. 1993. *Automatic Diagnosis of VLSI Digital Circuits Using Algorithmic Debugging*. Springer-Verlag, Berlin, 350–367. DOI: http://dx.doi.org/10.1007/BFb0019419

A. Lakhotia and L. Sterling. 1991. ProMiX: A prolog partial evaluation system. In *The Practice of Prolog*, L. Sterling (Ed.). The MIT Press, Cambridge, MA, Chapter 5, 137–179.

S. Liang. 1999. *Java Native Interface: Programmer's Guide and Reference (1st ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

J. W. Lloyd. 1987a. Declarative error diagnosis. *New Gen. Comput.* 5, 2 (1987), 133–154.

J. W. Lloyd. 1987b. *Foundations of Logic Programming* (2nd ed.). Springer-Verlag, Berlin.

F. López-Fraguas and J. Sánchez-Hernández. 1999. TOY: A multiparadigm declarative system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*. Lecture Notes in Computer Science, Vol. 1631. Springer-Verlag, Berlin, 244–247.

W. Lux. 2006. Münster Curry User's guide (Release 0.9.10 of May 10, 2006). Retrieved from http://danae.uni-muenster.de/~lux/curry/user.pdf.

W. Lux. 2008. Declarative debugging meets the world. *Electr. Not. Theoret. Comput. Sci.* 216 (2008), 65–77.

I. MacLarty. 2005. *Practical Declarative Debugging of Mercury Programs*. Ph.D. Dissertation. Department of Computer Science and Software Engineering, The University of Melbourne.

M. Maeji and T. Kanamori. 1987. *Top-Down Zooming Diagnosis of Logic Programs*. Technical Report TR-290. ICOT, Japan.

J. Naganuma, T. Ogura, and T. Hoshino. 1994. High-level design validation using algorithmic debugging. In *Proceedings of the European Conference on Design Automation (EDAC'94) and the European Test Conference (ETC'94), The*

European Event in ASIC Design (EUROASIC'94), Robert Werner (Ed.). IEEE Computer Society, New York, NY, 474–480. DOI: http://dx.doi.org/10.1109/EDTC.1994.326833

L. Naish. 1992a. Declarative debugging of lazy functional programs. In *Proceedings of the Workshop on Logic Programming Environments*. Case Western Reserver University, Cleveland, 29–34.

L. Naish. 1992b. Declarative diagnosis of missing answers. *New Gen. Comput.* 10, 3 (June 1992), 255–285.

L. Naish. 1997a. A declarative debugging scheme. *J. Funct. Logic Program.* 1997, 3 (1997), 1–27.

L. Naish. 1997b. A three-valued declarative debugging scheme. In *Proceedings of the Workshop on Logic Programming Environments (LPE'97)*. The MIT Press, Cambridge, MA, 1–12.

L. Naish, P. W. Dart, and J. Zobel. June 1989. *The NU-Prolog Debugging Environment*. Technical Report 88/31. Department of Computer Science, University of Melbourne, Lisboa, Portugal. 521–536.

H. Nilsson. 1998. *Declarative Debugging for Lazy Functional Languages*. Ph.D. Dissertation. Linköping, Sweden.

H. Nilsson. 2001. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *J. Funct. Program.* 11, 6 (2001), 629–671. DOI: http://dx.doi.org/10.1017/S095679680100418X

H. Nilsson and P. Fritzson. 1992. Algorithmic debugging for lazy functional languages. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, Maurice Bruynoogheand Martin Wirsing (Eds.). Springer-Verlag, Berlin, 385–399.

H. Nilsson and P. Fritzson. 1994. Algorithmic debugging for lazy functional languages. *J. Funct. Program.* 4, 3 (1994), 337–370.

H. Nilsson and J. Sparud. 1997. The evaluation dependence tree as a basis for lazy functional debugging. *Automat. Softw. Eng.* 4, 2 (1997), 121–150.

T. J. Ostrand and M. J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* 31, 6 (June 1988), 676–686. DOI: http://dx.doi.org/10.1145/62959.62964

L. M. Pereira. 1986. Rational debugging in logic programming. In *Proceedings of the 3rd International Conference on Logic Programming*. Lecture Notes in Computer Science, Vol. 225. Springer-Verlag, Berlin, 203–210.

B. Pope. 1998. *Buddha: A Declarative Debugger for Haskell*. Technical Report 98/12. Department of Computer Science and Software Engineering, The University of Melbourne.

B. Pope. 2006. *A Declarative Debugger for Haskell*. Ph.D. Dissertation. The University of Melbourne, Australia.

B. Pope and L. Naish. 2003. A program transformation for debugging haskell 98. In *Proceedings of the 26th Australasian Computer Science Conference (ACSC'03)*. Conferences in Research and Practice in Information Technology, Vol. 16. ACS, Darlinghurst, Australia, 227–236.

R. Ramakrishnan and J. D. Ullman. 1995. A survey of deductive database systems. *J. Logic Program.* 23, 2 (1995), 125–149.

A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. 2012. Declarative debugging of rewriting logic specifications. *J. Logic Algebr. Program.* 81, 7–8 (2012), 851–897.

L. De Rose, A. Gontarek, A. Vose, R. Moench, D. Abramson, M. Ngoc Dinh, and C. Jin. 2015. Relative debugging for a highly parallel hybrid computer system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, J. Kernand J. S. Vetter (Eds.). ACM, New York, NY, 63:1–63:12. DOI: http://dx.doi.org/10.1145/2807591.2807605

P. Runeson. 2006. A survey of unit testing practices. *IEEE Softw.* 23, 4 (July 2006), 22–29. DOI: http://dx.doi.org/10.1109/MS.2006.91

F. Russo and M. Sancassani. 1992. A declarative debugging environment for DATALOG. In *Proceedings of the 1st Russian Conference on Logic Programming*. Springer-Verlag, Berlin, 433–441.

F. Sáenz-Pérez. 2011. DES: A deductive database system. *Electron. Not. Theor. Comput. Sci.* 271 (March 2011), 63–78.

Herbert Schildt. 2014. *Java: The Complete Reference*. (9th ed.). McGraw–Hill Education, New York, NY.

N. Shahmehri and P. Fritzson. 1990. Algorithmic debugging for imperative languages with side-effects. In *Proceedings of the International Workshop on Compiler Construction*. Springer-Verlag, Berlin, 226–227.

E. Y. Shapiro. 1982a. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA.

E. Y. Shapiro. 1982b. Algorithmic program diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*. ACM, New York, NY, 299–308.

O. Shmueli and S. Tsur. 1991. Logical diagnosis of LDL programs. *New Gener. Comput.* 9, 3 (1991), 277.

J. Silva. 2006. Three new algorithmic debugging strategies. In *Proceedings of the VI Jornadas de Programación y Lenguajes (PROLE'06)*. 243–252.

J. Silva. 2007. A comparative study of algorithmic debugging strategies. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*. Lecture Notes in Computer Science, Vol. 4407. Springer-Verlag, Berlin, 143–159.

J. Silva. 2011. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.* 42, 11 (2011), 976–991.

J. Silva and O. Chitil. 2006. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*. ACM Press, New York, NY, 157–166.

J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. de Carvalho, and J. Gama. 2013. Data stream clustering: A survey. *Comput. Surv.* 46, 1, Article 13 (July 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2522968.2522981

R. Sosic and D. Abramson. 1997. Guard: A relative debugger. *Softw. Pract. Exp.* 27, 2 (1997), 185–206. DOI: http://dx.doi.org/10.1002/(SICI)1097-024X(199702)27:2⟨185::AID-SPE79⟩3.0.CO;2-D

L. Sterling and E. Shapiro. 1986. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, Cambridge, MA.

P. Kambam Sugavanam. 2013. *Debugging Framework for Attribute Grammars*. Ph.D. Dissertation. University of Minnesota.

S. Tamarit, A. Riesco, E. Martin-Martin, and R. Caballero. 2016. Debugging meets testing in erlang. In *Proceedings of the 10th International Conference on Tests and Proofs (TAP'16)*. Lecture Notes in Computer Science, Bernhard K. Aichernigand Carlo A. Furia (Eds.), Vol. 9762. Springer-Verlag, Berlin, 171–180.

A. Tessier and G. Ferrand. 2000. Declarative diagnosis in the CLP scheme. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski (Eds.). Springer-Verlag, Berlin, 151–174.

F. Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3 (1995), 121–189.

C. Wieland. 1990. Two explanation facilities for the deductive database management system DeDEx. In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90)*. Eidgenössische Technische Hochschule Zürich, Zürich, 189–203.

D. Wlodzimierz, S. Nadjm-Tehrani, and J. Maluszynski. 1988. The use of assertions in algorithmic debugging. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*. MIT Press Cambridge, MA, 573–581.

C. Zinn. 2013. Algorithmic debugging for intelligent tutoring: How to use multiple models and improve diagnosis. In *Proceedings of the 37th Annual German Conference on Advances in Artificial Intelligence (KI'13)*. Lecture Notes in Computer Science, Vol. 8077. Springer-Verlag, Berlin, 272–283. DOI: http://dx.doi.org/10.1007/978-3-642-40942-4_24

C. Zinn. 2014. Algorithmic debugging and literate programming to generate feedback in intelligent tutoring systems. In *Proceedings of the 37th Annual German Conference on Advances in Artificial Intelligence (KI'14)*. Lecture Notes in Computer Science, C. Lutzand M. Thielscher (Eds.), Vol. 8736. Springer-Verlag, Berlin, 37–48. DOI: http://dx.doi.org/10.1007/978-3-319-11206-0_4

Received February 2017; revised June 2017; accepted June 2017