TLP **19** (2): 290–316, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068418000492

*Debugging Non-ground ASP Programs: Technique and Graphical Tools**

CARMINE DODARO
DIBRIS, University of Genova, Genova, Italy
(*e-mail:* dodaro@dibris.unige.it)

PHILIP GASTEIGER
Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria
(*e-mail:* philip.gasteiger@gmail.com)

KRISTIAN REALE and FRANCESCO RICCA
Department of Mathematics and Computer Science, University of Calabria, Rende, Italy
(*e-mails:* reale@mat.unical.it, ricca@mat.unical.it)

KONSTANTIN SCHEKOTIHIN
Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria
(*e-mail:* konstantin.schekotihin@aau.at)

submitted 29 September 2017; revised 28 September 2018; accepted 28 September 2018

Abstract

Answer set programming (ASP) is one of the major declarative programming paradigms in the area of logic programming and non-monotonic reasoning. Despite that ASP features a simple syntax and an intuitive semantics, errors are common during the development of ASP programs. In this paper we propose a novel debugging approach allowing for interactive localization of bugs in non-ground programs. The new approach points the user directly to a set of non-ground rules involved in the bug, which might be refined (up to the point in which the bug is easily identified) by asking the programmer a sequence of questions on an expected answer set. The approach has been implemented on top of the ASP solver WASP. The resulting debugger has been complemented by a user-friendly graphical interface, and integrated in ASPIDE, a rich integrated development environment (IDE) for answer set programs. In addition, an empirical analysis shows that the new debugger is not affected by the grounding blowup limiting the application of previous approaches based on meta-programming.

KEYWORDS: answer set programming, debugging, graphical user interface

1 Introduction

Answer set programming (ASP) (Brewka *et al.* 2011; Gelfond and Lifschitz 1991) is a declarative programming paradigm proposed in the area of logic programming and non-monotonic reasoning. ASP features an expressive language that can be used to

* This paper includes parts and significantly extends our previous work (Dodaro *et al.* 2015). The authors are grateful to Marc Deneker and Ingmar Dasseville for the fruitful discussions about debugging for logic programs and FO(ID) theories, and in particular for the useful suggestion improving the handling of bugs caused by atoms missing a supporting rule. The authors are also grateful to Roland Kaminski for providing GRINGO without simplifications.

model computational problems of comparatively high complexity (Eiter *et al.* 1997) often in a rather compact way. The availability of high-performance implementations (Gebser *et al.* 2015; Lierler *et al.* 2016; Calimeri *et al.* 2016; Gebser *et al.* 2016) made ASP a valuable tool for developing complex applications in several research areas (Erdem *et al.* 2016), including Artificial Intelligence (Balduccini *et al.* 2001; Aschinger *et al.* 2011; Dodaro *et al.* 2015; Abseher *et al.* 2016; Dodaro *et al.* 2016), Hydroinformatics (Gavanelli *et al.* 2015), Nurse Scheduling (Alviano *et al.* 2017), and Bioinformatics (Erdem and Öztok 2015; Koponen *et al.* 2015; Gebser *et al.* 2011), to mention a few. Especially the development of real-world applications outlined the advantages of ASP from a software engineering viewpoint. Namely, ASP programs are flexible, compact, extensible, and easy to maintain (Grasso *et al.* 2011; Grasso *et al.* 2009).

Although the basic syntax of ASP is not particularly difficult, one of the most tedious and time-consuming programming tasks is the identification of (even trivial) faults in a program. For this reason, several methodologies and tools have been proposed in the last few years for debugging ASP programs (Brain and De Vos 2005; Pontelli *et al.* 2009; Oetsch *et al.* 2011; Gebser *et al.* 2008; Oetsch *et al.* 2010; Shchekotykhin 2015) with the goal of making the process of developing logic programs more rapid and comfortable. Given a faulty ASP program Π and a set of atoms I representing an interpretation of the program, these approaches find an explanation why I *is not an answer set* of Π.

The most prominent debugging approaches (Gebser *et al.* 2008; Oetsch *et al.* 2010) apply the notion of meta-programming that uses ASP itself to debug a faulty ASP program. The basic idea of the meta-programming method is to convert the inputs into a program over a meta-language – a reified program – and then execute it together with a debugging program. The latter finds causes of a fault, where each cause is encoded by specific atoms in an answer set of the debugging program. However, reification-based debuggers have some issues that may make them either difficult to apply or even inapplicable in some cases. The main issue, also observed in Oetsch *et al.* (2010), is related to the computation of answer sets of the debugging program. Namely, the grounding step of the solving process might produce ground instantiations of the debugging program, which sizes are exponentially larger than the input. This problem is intrinsic in the meta-programming approach, as it requires the ground debugging program to comprise a set of ground rules encoding all possible explanations of faults in the input program. Moreover, even if answer sets of the reified program can be computed, their number can overwhelm a user, who has to analyze all explanations manually in order to find a true cause of the problem. A recent approach suggested in Shchekotykhin (2015) allows for a partial resolution of this issue, but it is applicable only to ground programs, which often contains a large number of rules and, consequently, are hard to understand and to debug.

In this paper we propose a novel debugging approach allowing for interactive localization of bugs in non-ground ASP programs. The new approach points the user directly to a set of non-ground rules involved in the bug, which might be refined (up to the point in which the bug is easily identified) by asking the programmer a sequence of questions on an expected answer set. Roughly, the suggested approach can be described as follows: First, given a non-ground program Π the debugger generates a debugging program Δ_Π by adding marker atoms used later to match results of the debugger with rules of Π; Next, the debugging program is grounded and passed, together with (the second input) an interpretation I, to a specifically modified version of an ASP solver that determines the set

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

of rules that are possible reasons for the bug. As previous approaches, our debugger allows for a uniform treatment of *over-constrained* and *missing support* faults (see Section 3). Since the number of reasons might be large (up to covering the entire program), we help the user to find the *guilty rules* by applying an automated refining technique. In particular, the debugger automatically generates a sequence of queries, requiring the user to answer whether a set of literals must be included or not in an expected answer set. This additional knowledge is then injected in the system and the process is repeated up to a point in which the (non-ground) rules causing a fault can easily be identified.

The approach has been implemented in the DWASP debugger that combines the grounder GRINGO (Gebser *et al.* 2011) with an extension of the ASP solver WASP (Alviano *et al.* 2015). The resulting implementation can be used via command-line interface. In order to further ease the task of debugging logic programs, and appeal to those users that prefer graphical interfaces, we also developed a graphical user interface for the DWASP debugger, called DWASP-GUI, and a plugin connector for the integrated development environment (IDE) ASPIDE. An important reason to choose ASPIDE over other ASP IDEs, like Sealion (Busoniu *et al.* 2013), is its richer support for test-driven development of ASP programs (Febbraro *et al.* 2011). The test-driven development process (Fraser *et al.* 2003) requires the repetition of a very short development cycle in which requirements are encoded as specific test cases (assessing a possibly small unit of the program), and the program is assessed against tests, and possibly fixed or improved. The cycle is then repeated to push forward the functionality, until the program satisfies all the requirements. The rapid identification of the cause of a failing test case is fundamental for test-driven development platforms. In ASPIDE a user can naturally define test cases comprising inputs and expected (non-)outputs of a solver when applied to compute answer sets of a developed ASP program. Services of the IDE allow a user to execute the test cases and to generate a report of the results. Thus, for a reported failed test case, the plug-in connector automatically generates and forwards to DWASP all required inputs, configures, and executes of debugging tasks relevant to the studied test case. In this way the DWASP debugger synergistically works with the test-driven framework of ASPIDE resulting in a more complete test-driven development environment. Overall, the combination provides an intuitive user-experience with ASPIDE similar to the one of modern IDEs for software development with imperative languages.

To summarize, the paper makes the following contributions[1]:

1. We present an interactive and efficient debugging technique for non-ground ASP programs (see Section 3).
2. We implement a tool, called DWASP, for debugging *non-ground* ASP programs supporting all syntax of the latest ASP-Core (see Section 4.1).
3. We suggest a new graphical debugging interface for ASP programs, called DWASP-GUI, based on DWASP that improves the user-experience of the debugger (see Section 4.2).
4. We integrate DWASP-GUI over a new plug-in connector with ASPIDE (see Section 4.3).

[1] This paper is an extended and improved version of Dodaro *et al.* (2015) featuring the following improvements: (i) we provide a new formal description of the debugging approach and (ii) we prove some of its formal properties; (iii) we extend the approach with "missing support" faults; (iv) we implement the DWASP-GUI; (v) we extend ASPIDE with the new debugger; and (vi) we perform a usability testing assessment on students of a course on ASP.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

5. We compare our tool with state-of-the-art debuggers based on meta-programming approaches (Gebser *et al.* 2008; Oetsch *et al.* 2010) and show that our approach is basically unaffected by the combinatorial blow-up which limits the performance of meta-programming approaches [cfr. (Oetsch *et al.* 2010)] (see Section 5).
6. We report the results of an assessment of usability and appreciation of the debugger obtained by running a user-experience experiment involving students of a course on ASP (see Section 6).

2 Answer set programming

In this section we overview ASP focusing on preliminary notions that are required for describing the debugging approach implemented in DWASP. The reader is referred to Baral (2010) and Gebser *et al.* (2012) for a more comprehensive presentation of ASP.

Syntax. A program Π is a finite set of rules of the form

$$a_1 \vee \ldots \vee a_m \leftarrow l_1, \ldots, l_n, \tag{1}$$

where $a_1, \ldots, a_m$ are atoms and $l_1, \ldots, l_n$ are literals for $m, n \geq 0$. In particular, an *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are *terms*. Terms are alphanumeric strings and are distinguished in variables and constants. According to the Prolog's convention, only variables start with an uppercase letter. A *literal* is an atom a_i (positive) or its negation $\sim a_i$ (negative), where $\sim$ denotes the *negation as failure*. An atom, literal, or rule is called *ground* if it contains no variable. The complement of a literal l is denoted by $\bar{l}$. In particular, given atom a it holds that $\overline{a} = \sim a$ and $\overline{\sim a} = a$. Moreover, the complement for a set of literals L is $\overline{L} := \{\bar{l} \mid l \in L\}$. Given a rule r of the form (1), the set of atoms $H(r) = \{a_1, \ldots, a_m\}$ is called *head* and the set of literals $B(r) = \{l_1, \ldots, l_n\}$ is called *body*. Moreover, $B(r)$ can be partitioned into the sets $B^+(r)$ and $B^-(r)$ comprising the positive and negative body literals, respectively. A rule r is called *fact* if $|H(r)| = 1$ and $B(r) = \emptyset$ and *constraint* if $H(r) = \emptyset$. Every rule $r \in \Pi$ must be *safe*, that is, each variable of r must occur in at least one positive literal of $B^+(r)$. In the following, we will also use *choice rules* of the form $\{a\}$, where a is a ground atom. A choice rule $\{a\}$ is hereafter considered as a syntactic shortcut for the rule $a \vee a_{\mathrm{F}} \leftarrow$, where a_{F} is a fresh new atom not appearing elsewhere in the program.

Semantics. Let Π be an ASP program, the Herbrand Universe U_Π, and the Herbrand base B_Π are defined as usual. The semantics of an ASP program is given in terms of the answer sets of its ground instantiation. The ground instantiation of Π, denoted by Π^G, is the ground program obtained by properly substituting all variables occurring in rules from Π with elements of U_Π. An *interpretation* is a set of ground atoms $I \subseteq B_\Pi$. Relation $\models$ is inductively defined as follows: for $a \in B_\Pi$, $I \models a$ if $a \in I$, otherwise $I \not\models a$; $I \models \sim a$ if $I \not\models a$; for a set of atoms S, $I \models S$ if $I \models l$ for all $l \in S$, otherwise $I \not\models S$; for a rule $r \in \Pi^G$, $I \models r$ if $I \cap H(r) \neq \emptyset$ whenever $I \models B(r)$; for a program Π^G, $I \models \Pi^G$ if $I \models r$ for all $r \in \Pi^G$. I is a *model* of Π^G if $I \models \Pi^G$.

The *reduct* Π^G_I of a program Π^G with respect to an interpretation I is obtained from Π^G as follows: (i) any rule r such that $I \not\models B^-(r)$ is removed; (ii) any negated literal l such that $I \not\models l$ is removed from the body of the remaining rules. An interpretation I is an *answer set* (stable model) of a program Π^G if $I \models \Pi^G$, and there is no $J \subset I$ such

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

that $J \models \Pi^G_I$. The set of all answer sets of Π is denoted by $\mathrm{AS}(\Pi)$. Π is *incoherent*, if $\mathrm{AS}(\Pi) = \emptyset$, and *coherent* otherwise.

Support. Given a model I for a ground program Π, we say that a ground atom $a \in I$ is *supported* with respect to I if there exists a *supporting* rule $r \in \Pi^G$ such that $I \models B(r)$, $I \models a$, and $I \not\models (H(r) \backslash \{a\})$. As it follows from the definition of the semantics given above, all atoms in an answer set I must be supported.

3 Debugging approach

In the following we present our approach to interactive localization of faults in non-ground ASP programs. In general, one can differentiate between syntactic and semantic faults which require specific methods for debugging them. The first type of faults is usually detected by parsers of ASP grounders, whereas semantic faults can only be observed by a user while analyzing answer sets returned by a solver. In order to detect faults of the second type many ASP users verify the correctness of a program by testing it on a sample instance, which is common for software development. In this case a user compares a (sub)set of all answer sets returned by a solver with expected solutions determined by hand. Therefore, often at least one answer set of the program for the sample instance is known to the user, otherwise it is impossible to understand that some program is buggy. That is, a bug is then revealed when the known answer set is not among the computed ones.

Definition 1 (*Buggy program*)
Let Π^c be the intended (correct) program that a user is going to formulate and $\mathrm{AS}(\Pi^c)$ be a set of its known answer sets. Then, a program Π is said to be *buggy* with respect to a program Π^c if there exists an answer set $A \in \mathrm{AS}(\Pi^c)$ such that $A \notin \mathrm{AS}(\Pi)$.

Note that by this definition our approach deals only with situations in which some answer set of the correct program is missing. The opposite problem – there is an answer set $A \in AS(\Pi)$ such that $A \notin AS(\Pi^c)$ – is not the focus of this paper.

Example 1 (*Buggy program*)
Consider the program Π' representing a (buggy) encoding for the graph coloring problem:

$$node(X) \leftarrow edge(X, Y)$$
$$node(X) \leftarrow edge(Y, X)$$
$$col(X, blue) \vee col(X, red) \vee col(X, green) \leftarrow node(X)$$
$$\leftarrow col(X, C_1), col(Y, C_2), edge(X, Y), X \neq Y, C_1 \neq C_2.$$

During the development of the encoding the user might create a simple graph, for example, considering the sample instance comprising two facts:

$$edge(1, 2) \leftarrow \qquad edge(2, 3) \leftarrow .$$

For this instance the user expects the assignment of the *blue* color to the nodes 1 and 3 as well as of the *red* color to the node 2 to be among the solutions. However, the corresponding answer set encoding this solution is missing due to a bug in the encoding. In particular, note that the condition $C_1 \neq C_2$ should be replaced by $C_1 = C_2$. ◁

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

The situation in which some solution is missing can be detected by means of testing, which is a common approach in software engineering aiming at identification and localization of faults in programs.

Definition 2 (*Test case*)
Let Π^c be the intended program, Π be a program, and B_Π be a Herbrand base of Π. A set of atoms $T \subseteq B_\Pi$ is a *test case* for a program Π iff there exists an answer set $A \in \mathrm{AS}(\Pi^c)$ such that $T \subseteq A$.

Definition 3 (*Test case failure*)
Given a program Π and a test case T, let $\Pi_T = \{\leftarrow \bar{l} \mid l \in T\}$, we say that T *fails* if $\Pi \cup \Pi_T$ is incoherent.

Assertions of a test case are modeled by constraints that force the asserted atoms to be in all answer sets. As a result, checking whether a test case T of a program Π passes or not is reduced to checking whether $\Pi \cup \Pi_T$ is coherent, as illustrated in Example 2.

Example 2 (*Failing test case*)
Consider the program Π' from Example 1 and the test case

$$T = \{col(1, blue), col(2, red), col(3, blue)\}.$$

The program Π'_T is composed by the constraints $\leftarrow \sim col(1, blue)$, $\leftarrow \sim col(2, red)$, and $\leftarrow \sim col(3, blue)$. Thus, T is failing since $\Pi' \cup \Pi'_T$ is incoherent. ◁

Whenever a test case fails, that is, the given program Π is buggy, the goal of a debugger is to find an explanation for this observation. However, in many cases it might be obvious to a user that some rules in Π are definitely correct and are not related to a fault, for example, facts defining the test instance or some simple rules. In such situations, the user might want to communicate this background knowledge to the debugger in order to exclude explanation candidates that are not explanations of the fault. In practice, this allows the debugger to stay focused on the fault and reduce its runtime.

Definition 4 (*Background knowledge*)
Given a program Π the *background knowledge* $\mathcal{B} \subset \Pi$ is a set of rules considered to be correct.

Example 3 (*Background knowledge*)
Consider the program Π' from Example 1 and assume that the user is sure that the sample instance is encoded correctly. Therefore, the background knowledge $\mathcal{B}'$ of Π' is composed by the facts $edge(1, 2) \leftarrow$ and $edge(2, 3) \leftarrow$. ◁

Note that while working on various industrial applications and developing this debugging approach, we found that it is advantageous to move facts of test instances to the background knowledge. This behavior is implemented as the default one in our debugger, unless the user provides a custom definition of the background knowledge.

In general, there are two possible causes for the incoherence of $\Pi \cup \Pi_T$ considered in the literature on ASP debugging: (1) over-constrained programs and (2) missing support. In the first case we would like to find and highlight a set of rules in $\Pi \backslash \mathcal{B}$ that erroneously constrain the set of all answer sets and eliminate the intended ones. If the problem is

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

due to the missing support, that is, none of the rules in $\Pi \backslash \mathcal{B}$ allow for derivation of some atoms in the intended answer set, then we would like to highlight the corresponding atoms in the test case.

Example 4 (*Errors detection*)
Consider the program Π' and the test case T from Example 2. A debugger should identify the buggy rule:

$$\leftarrow col(X, C_1), col(Y, C_2), edge(X, Y), X \neq Y, C_1 \neq C_2.$$

Indeed, given the constraints $\leftarrow \sim col(1, blue)$ and $\leftarrow \sim col(2, red)$ in Π'_T, the above constraint cannot be satisfied. In that case, the condition $C_1 \neq C_2$ should be replaced by $C_1 = C_2$. ◁

In our approach, fault identification is done by constructing a specific program that allows a solver to find rules and/or atoms explaining the fault. Note that this program is different from the one generated by meta-programming debuggers, since it uses no reification (see Sections 5 and 7 for more details). Instead, we only extend the buggy program in a way that allows the debugger to map its results back to the input program.

Definition 5 (*Debugging program*)
Let Π be a program, $\mathcal{B}$ be the background knowledge, and $id : (\Pi \backslash \mathcal{B}) \rightarrow \mathbb{N}$ be an assignment of unique identifiers to the non-background knowledge rules of Π. Then, given

1. the program $\Delta_\Pi^D = \{H(r) \leftarrow B(r) \cup \{\_debug(id(r), \vec{vars})\} \mid r \in (\Pi \backslash \mathcal{B})\}$, where $\_debug(id(r), \vec{vars})$ is a fresh atom and $\vec{vars}$ is a tuple with all variables of r and
2. the program $\Delta_\Pi^S = \{a \leftarrow \sim \_support(a) \mid a \in B_\Pi\}$, where $\_support(a)$ is a fresh atom called *supporting atom* of a,

the *debugging program* Δ_Π of Π is defined as $\Delta_\Pi = \Delta_\Pi^D \cup \Delta_\Pi^S \cup \mathcal{B}$.

Example 5 (*Debugging program*)
Consider the program Π' and the background knowledge $\mathcal{B}'$ from Example 3. The *debugging program* $\Delta_{\Pi'}$ is the following set of rules:

$$\begin{array}{lll}
\Delta_\Pi^D: & node(X) \leftarrow edge(X, Y), _debug(1, X, Y) & \\
& node(X) \leftarrow edge(Y, X), _debug(2, Y, X) & \\
& col(X, blue) \vee col(X, red) \vee col(X, green) \leftarrow node(X), _debug(3, X) & \\
& \leftarrow col(X, C_1),\ col(Y, C_2),\ edge(X, Y), & \\
& \qquad X \neq Y,\ C_1 \neq C_2,\ _debug(4, X, Y, C_1, C_2) & \\
\Delta_\Pi^S: & node(i) \leftarrow \sim _support(node(i)) & \forall\ i \in \{1, 2, 3\} \\
& edge(i, j) \leftarrow \sim _support(edge(i, j)) & \forall\ (i, j) \in \{(1, 2), (2, 3)\} \\
& col(n, c) \leftarrow \sim _support(col(n, c)) & \forall\ n \in \{1, 2, 3\}, \forall\ c \in \{blue, red, green\} \\
\mathcal{B}: & edge(1, 2) \leftarrow & edge(2, 3) \leftarrow
\end{array}$$

◁

Since atoms of the form $\_debug(id(r), \vec{vars})$ and $\_support(a)$ only appear in the body of the rules of Δ_Π, they are not supported. Therefore, Δ_Π is extended to provide a supporting rule for all atoms of this form.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Definition 6 (*Extended debugging program*)
Let Δ_Π be a debugging program, $\mathcal{A}^D = \{\_debug(id(r), v\vec{a}rs) \mid \_debug(id(r), v\vec{a}rs) \in B_{\Delta_\Pi}\}$ and $\mathcal{A}^S = \{\_support(a) \mid \_support(a) \in B_{\Delta_\Pi}\}$. Then, an *extended debugging program* Δ^*_Π is defined as $\Delta_\Pi \cup \{\{a\} \leftarrow \mid a \in (\mathcal{A}^D \cup \mathcal{A}^S)\}$, where $\{a\} \leftarrow$ denotes a choice rule (Simons et al. 2002).

Example 6 (*Extended debugging program*)
Consider $\Delta_{\Pi'}$ from Example 5. Given $\Delta_{\Pi'}$ an intelligent grounder would output a program that comprises only the ground rules derived from the background knowledge, namely $edge(1,2) \leftarrow$ and $edge(2,3) \leftarrow$. All other rules will be dropped because of the atoms over *_debug* and *_support* predicates.

The extended debugging program $\Delta^*_{\Pi'}$ comprises the following additional set of rules:

$$
\begin{array}{ll}
\{\_debug(1,i,j)\} \leftarrow & \forall\ (i,j) \in \{(1,2),(1,3)\} \\
\{\_debug(2,i,j)\} \leftarrow & \forall\ (i,j) \in \{(1,2),(1,3)\} \\
\{\_debug(3,i)\} \leftarrow & \forall\ i \in \{1,2,3\} \\
\{\_debug(4,1,2,c_1,c_2)\} \leftarrow & \forall\ c_1, c_2 \in \{blue, red, green\} \mid c_1 \neq c_2 \\
\{\_support(node(i))\} \leftarrow & \forall\ i \in \{1,2,3\} \\
\{\_support(edge(i,j))\} \leftarrow & \forall\ (i,j) \in \{(1,2),(1,3)\} \\
\{\_support(col(n,c))\} \leftarrow & \forall\ n \in \{1,2,3\}, \forall\ c \in \{blue, red, green\}
\end{array}
\quad \lhd .
$$

These rules provide the necessary support to the fresh body atoms of the rules in Δ^D_Π and Δ^S_Π thus disabling the simplifications of a grounder.

It is important to show that the extended debugging program preserves some properties of the original program. In particular, in the following it is shown that, under some conditions, the extended debugging program is coherent if and only if the original program is coherent.

Proposition 1
Let Π be a program, $\mathcal{B}$ a background knowledge, and T a test case. In addition, let $\Pi_\mathcal{A} = \{a \leftarrow \mid a \in (\mathcal{A}^D \cup \mathcal{A}^S)\}$. Then, a program $\Gamma_\Pi = \Delta^{'*}_\Pi \cup \Pi_T \cup \Pi_\mathcal{A}$ is coherent iff $\Pi \cup \Pi_T$ is coherent.

Proof sketch
The proof follows from the observation that the set of facts $\Pi_\mathcal{A}$ in the program Γ_Π reduces Δ^*_Π to Π. Namely, the set of rules $\{\{a\} \leftarrow \mid a \in (\mathcal{A}^D \cup \mathcal{A}^S)\}$ is trivially satisfied given $\Pi_\mathcal{A}$ and can be removed from consideration. Moreover, all atoms over *_debug* predicate must be valuated to true because of $\Pi_\mathcal{A}$ and can be removed from bodies of corresponding rules. Finally, none of the bodies of rules in Δ^S_Π are satisfied given $\Pi_\mathcal{A}$ and, therefore, these rules are also removed. □

Consequently, checking the correctness of a test case T of a program Π and background knowledge $\mathcal{B}$ can be done by verifying if Γ_Π is coherent. In case Γ_Π is incoherent, and so is the $\Pi \cup \Pi_T$, we can use Γ_Π to find the reason of the incoherence.

Definition 7 (*Reason of incoherence*)
Let Γ_Π be an incoherent program. A set of rules $\mathcal{R} \subseteq \Pi_\mathcal{A}$ is a reason of incoherence for Γ_Π if $(\Gamma_\Pi \backslash \Pi_\mathcal{A}) \cup \mathcal{R}$ is incoherent. A reason of incoherence $\mathcal{R}$ is *minimal* if there is no set of rules $\mathcal{R}' \subset \mathcal{R}$ such that $\mathcal{R}'$ is reason of incoherence for Γ_Π.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Example 7 (*Reason of incoherence*)
Consider the extended debugging program $\Delta^*_{\Pi'}$ from Example 6 and the following test case:

$$T = \{col(1, blue), col(2, red), col(3, blue)\}.$$

Note that $\Gamma_{\Pi'} \backslash \Pi'_{\mathcal{A}}$ is coherent, whereas Γ_{Π} is incoherent. Thus, a (trivial) reason of incoherence would be the whole set $\Pi'_{\mathcal{A}}$. There are two minimal reasons of incoherence, that is, $\mathcal{R}_1 = \{\_debug(4, 1, 2, blue, red) \leftarrow\}$ and $\mathcal{R}_2 = \{\_debug(4, 2, 3, blue, red) \leftarrow\}$. Clearly, when atoms $col(1, blue)$ and $col(2, red)$ are true, the rule

$$\leftarrow col(X, C_1), col(Y, C_2), edge(X, Y), X \neq Y, C_1 \neq C_2$$

with the instantiation $X = 1$, $Y = 2$, $C_1 = blue$, and $C_2 = red$ is violated, thus the atom $\_debug(4, 1, 2, blue, red)$ cannot be true. Moreover, note that both reasons originate from the same non-ground rule and are due to the symmetry of substitutions. ◁

One important property of reasons of incoherence is their monotonicity, that is, if a set of rules $\mathcal{R} \subseteq \Pi_{\mathcal{A}}$ is a reason of incoherence, then all supersets $\mathcal{R}_1 \subseteq \Pi_{\mathcal{A}}$ of $\mathcal{R}$ are also reasons of incoherence.

Theorem 1 (*Monotonicity*)
Let Π be a program, T a test case, $\mathcal{B}$ a background knowledge, and Δ^*_{Π} an extended debugging program over Π and $\mathcal{B}$. Let $\Gamma_{\Pi} = \Delta^*_{\Pi} \cup \Pi_T \cup \Pi_{\mathcal{A}}$ be an incoherent program and $\mathcal{R} \subseteq \Pi_{\mathcal{A}}$ be a reason of incoherence, that is, $(\Gamma_{\Pi} \backslash \Pi_{\mathcal{A}}) \cup \mathcal{R}$ is incoherent by definition. Then, any set of rules $\mathcal{R}_1$, such that $\mathcal{R} \subset \mathcal{R}_1 \subseteq \Pi_{\mathcal{A}}$, is a reason of incoherence.

Proof
Let $\mathcal{P} = \Gamma_{\Pi} \backslash \Pi_{\mathcal{A}}$. Suppose that $\mathcal{P} \cup \mathcal{R}_1$ is coherent and M_1 is an answer set. We will prove that M_1 is an answer set of $\mathcal{P} \cup \mathcal{R}$. Therefore, we have a contradiction.

Let $\mathcal{R}_2 = \mathcal{R}_1 \backslash \mathcal{R}$ and let $\mathcal{A}_{\mathcal{R}_2} = \{a \mid a \leftarrow \in \mathcal{R}_2\}$. For each atom $a \in \mathcal{A}_{\mathcal{R}_2}$ [i.e. of the form $\_debug(\cdot)$ or $\_support(\cdot)$] there is a choice rule of the form $\{a\} \leftarrow \in \mathcal{P}$. Therefore, since $\{a\} \leftarrow$ is the only rule containing a in the head, the following property holds:

$$\begin{gathered}
\mathrm{AS}(\mathcal{P} \cup \mathcal{R}) \\
= \\
\mathrm{AS}(\mathcal{P} \cup \mathcal{R} \cup \{\{a\} \leftarrow \mid a \in \mathcal{A}_{\mathcal{R}_2}\}) \\
\supseteq \\
\mathrm{AS}(\mathcal{P} \cup \mathcal{R} \cup \{a \leftarrow \mid a \in \mathcal{A}_{\mathcal{R}_2}\}) \\
= \\
\mathrm{AS}(\mathcal{P} \cup \mathcal{R} \cup \mathcal{R}_2) \\
= \\
\mathrm{AS}(\mathcal{P} \cup \mathcal{R}_1).
\end{gathered}$$

Therefore, $\mathrm{AS}(\mathcal{P} \cup \mathcal{R}) \supseteq \mathrm{AS}(\mathcal{P} \cup \mathcal{R}_1)$. Thus, if M_1 is an answer set of $\mathcal{P} \cup \mathcal{R}_1$, then M_1 is an answer set of $\mathcal{P} \cup \mathcal{R}$, which is impossible since by definition $\mathcal{P} \cup \mathcal{R}$ is incoherent. Consequently, $\mathcal{P} \cup \mathcal{R}_1$ is also incoherent. □

Note that there are multiple ways to prove Theorem 1. For instance, we can use the definition of reduct given in Section 2. The idea of the proof is based on the fact that for any test case T the program Π_T defines a set of possible interpretations by constraining the truth assignments of atoms in T. According to the definition of the reduct, for each

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

interpretation I there is only one reduct corresponding to it. Since bodies of all rules in the reduct are positive, that is, comprise no negative literals, the consequence relation is monotonic. Therefore, we can always find at least one minimal reason of incoherence for every reduct and, consequently, for every interpretation allowed by the given test case.

Intuitively, a reason of incoherence represents a set of rules that makes the program incoherent. If the reason is minimal, removing one of those rules from the program makes it coherent. Thus, debugging an incoherent program can be reduced to the process of finding a minimal reason of incoherence, fix it and then reiterate the process until all reasons have been analyzed. However, in some cases a reason of incoherence might contain a large number of rules, thus making it infeasible to find the buggy rule among them. Therefore, we aim at reducing the reasons of incoherence by querying the user on the atoms that must belong to the intended answer set.

Example 8 (*Buggy encoding*)
Consider the following program Π'':

$$a \leftarrow c \qquad b \leftarrow \sim c \qquad c \leftarrow \sim b \qquad \leftarrow c, \sim b$$

and the test case $T = \{a\}$. The program $\Delta_{\Pi''}$ obtained from Π'' is the following:

$$\begin{array}{llll} a \leftarrow c, \_debug(1) & b \leftarrow \sim c, \_debug(2) & c \leftarrow \sim b, \_debug(3) & \leftarrow c, \sim b, \_debug(4) \\ a \leftarrow \sim \_support(a) & b \leftarrow \sim \_support(b) & c \leftarrow \sim \_support(c). & \end{array}$$

In this case, $\mathcal{R} = \{\_debug(4) \leftarrow, \_support(a) \leftarrow, \_support(b) \leftarrow\}$ is a minimal reason of incoherence of the program $\Gamma_{\Pi''}$. The intuitive meaning is that when the rule $\leftarrow c, \sim b$ is in the program the test case fails because a and b cannot be supported. Thus, the source of the error might be the rule $\leftarrow c, \sim b$ or one of the rules containing a and b in the head. The idea is to query the user to reduce the possible source of errors. ◁

Definition 8 (*Query*)
Let Γ_Π be an incoherent program , let T be a test case, and let $\mathcal{R}$ be a minimal reason of incoherence. A *query* is an atom $q \in B_\Pi \backslash T$. Let $\Gamma^*_\Pi = (\Gamma_\Pi \backslash \Pi_\mathcal{A}) \cup \mathcal{R}$, we define

$$Q^+(q) = \bigcup_{r \in \mathcal{R}} \{I \mid q \in I, I \in AS(\Gamma^*_\Pi \backslash \{r\})\}$$
and
$$Q^-(q) = \bigcup_{r \in \mathcal{R}} \{I \mid q \notin I, I \in AS(\Gamma^*_\Pi \backslash \{r\})\}.$$

Note that if $\mathcal{R}$ is minimal, then $\Gamma^*_\Pi \backslash \{r\}$ is coherent, for each $r \in \mathcal{R}$. For a query atom q, the set $Q^+(q)$ contains all answer sets in which the query atom q is true, whereas $Q^-(q)$ contains all answer sets in which q is false. Such sets are used to discriminate which atom is selected as query atom. The user then should confirm whether the query atom q is or not in the intended answer set.

Example 9 (*Query*)
Consider the program Π'' from Example 8 and the minimal reason of incoherence $\mathcal{R} = \{\_debug(4) \leftarrow, \_support(a) \leftarrow, \_support(b) \leftarrow\}$. A query atom is one of b and c. The program $\Gamma''^*_\Pi \backslash \{\_debug(4) \leftarrow\}$ admits the following answer sets:

$$I_1 = \{a, c, \_support(a), \_support(b), \_debug(1), \_debug(3)\},$$
$$I_2 = \{a, c, \_support(a), \_support(b), \_debug(1), \_debug(2), \_debug(3)\},$$
$$I_3 = \{a, c, \_support(a), \_support(b), \_support(c), \_debug(1), \_debug(3)\},$$

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

$I_4 = \{a, c, \_support(a), \_support(b), \_support(c), \_debug(1), \_debug(2), \_debug(3)\},$
$I_5 = \{a, c, \_support(a), \_support(b), \_debug(1)\},$
$I_6 = \{a, c, \_support(a), \_support(b), \_debug(1), \_debug(2)\}.$

Next, the program $\Gamma''^{*}_{\Pi} \backslash \{\_support(a)\}$ admits the following answer sets:

$I_7 = \{a, b, \_support(b), \_support(c), \_debug(2), \_debug(4)\},$
$I_8 = \{a, b, \_support(b), \_support(c), \_debug(2), \_debug(3), \_debug(4)\},$
$I_9 = \{a, b, \_support(b), \_support(c), \_debug(1), \_debug(2), \_debug(4)\},$
$I_{10} = \{a, b, \_support(b), \_support(c), \_debug(1), \_debug(2), \_debug(3), \_debug(4)\},$
$I_{11} = \{a, \_support(b), \_support(c), \_debug(4)\},$
$I_{12} = \{a, \_support(b), \_support(c), \_debug(1), \_debug(4)\}.$

Finally, the program $\Gamma''^{*}_{\Pi} \backslash \{\_support(b)\}$ admits the following answer sets:

$I_{13} = \{a, b, c, \_support(a), \_debug(1), \_debug(4)\},$
$I_{14} = \{a, b, c, \_support(a), \_debug(1), \_debug(2), \_debug(4)\},$
$I_{15} = \{a, b, c, \_support(a), \_debug(1), \_debug(3), \_debug(4)\},$
$I_{16} = \{a, b, c, \_support(a), \_debug(1), \_debug(2), \_debug(3), \_debug(4)\}.$

Then, $Q^+(b) = \{I_7, \ldots, I_{10}, I_{13}, \ldots, I_{16}\}$ and $Q^-(b) = \{I_1, \ldots, I_6, I_{11}, I_{12}\}$, while $Q^+(c) = \{I_1, \ldots, I_6, I_{13}, \ldots, I_{16}\}$ and $Q^-(c) = \{I_7, \ldots, I_{12}\}$. ◁

After computing the sets $Q^+(p)$ and $Q^-(p)$ for all atoms p, the idea is to select a query atom in a way that, regardless the answer to the query, the number of possible fixes is cut in half, that is, the atom q such that the absolute value of $|Q^+(q)| - |Q^-(q)|$ is minimum. When the atom q is selected, the user considers whether q to be true in the expected answer set. If q must be true, then $\leftarrow \sim q$ is added to the extended debugging program, otherwise $\leftarrow q$ is added.

Example 10 (*Query session*)
Let us continue Example 8. The atom b is selected as query atom, since $|Q^+(b)| - |Q^-(b)| = 0$. When the query b is selected, the user considers whether b to be true in the expected answer set. Assume the user selects b to be true and $\leftarrow \sim b$ is added to $\Delta^*_{\Pi''}$. For the new version of the extended debugging program we compute the new minimal reason of incoherence $\mathcal{R} = \{\_support(a) \leftarrow, \_support(b) \leftarrow\}$. If the user answers the next only possible query c with false, the $\Delta^*_{\Pi''}$ is extended with $\leftarrow c$. Thus, the newly computed reason of incoherence comprises only one rule $\mathcal{R} = \{\_support(a) \leftarrow\}$.

4 A debugger based on DWASP

In this section, a new graphical debugger based on DWASP, called DWASP-GUI, and its integration in ASPIDE (Febbraro *et al.* 2011) are presented by running an example.

4.1 The DWASP *debugger*

Our implementation of the DWASP debugger consists of two components: the debugging grounder GRINGO-WRAPPER and DWASP. Figure 1 illustrates the interaction of both components to debug a program Π. First, the program Π is read by GRINGO-WRAPPER

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

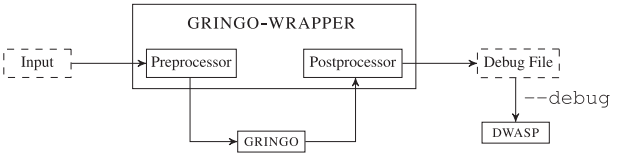


Fig. 1. Interaction of GRINGO-WRAPPER and DWASP in debugging mode.

from either the standard input or several input files. The debugging grounder internally transforms the input program, passes the result to an ASP grounder, and outputs the ground debugging program to the standard output, which is then processed by DWASP to start the interactive debugging session. In general, GRINGO-WRAPPER supports any `lparse`-compatible grounder; however, in our implementation we use GRINGO (Gebser *et al.* 2011).

Grounding with GRINGO-WRAPPER. The task of GRINGO-WRAPPER is to obtain the grounded debugging program given an input program Π and some test case T. First, Π is translated to the extended debugging program Δ^*_Π, as described in Definition 6 and T is translated into Π_T. In case a user does not provide the background knowledge, by default, all facts of Π are assumed to be correct, that is, the background knowledge $\mathcal{B}$ comprises all facts of Π. After this transformation, GRINGO is used to obtain the ground version of $\Delta^*_\Pi \cup \Pi_T$. However, modern grounders perform several optimizations during grounding, such as deriving new facts from normal rules. Although these optimizations potentially decrease the time required by the solver, they are counterproductive when debugging a logic program because wrong facts could be derived from faulty rules. Moreover, a grounder might remove entire non-ground rules that are missing support. In this case, GRINGO-WRAPPER issues a warning message that highlights the rules that were removed by the grounder.

There are a number of ways to avoid the removal of atoms or simplification of rules done by grounders. One can use `--keep-facts` option of GRINGO (since version 4.5.4) or use the following workaround implemented in GRINGO-WRAPPER: first, the wrapper performs a call to the grounder and analyzes the produced *atoms table* of the `lparse` format, that is, a list of ground atoms occurring in the ground program. Then, for each atom p the choice rule $\{p\}$ is added to the original program and the grounder is called again. These additional rules are removed in a postprocessing step.

Debugging session with DWASP. DWASP is a specialized variant of WASP (Alviano *et al.* 2015), a state-of-the-art ASP solver. WASP is based on a CDCL-like backtracking algorithm (Silva and Sakallah 1999), featuring the so-called *incremental interface* (Alviano *et al.* 2015). In particular, WASP can take as input a ground ASP program Π and a set of atoms A, called *assumptions*, and computes either an answer set $I \supseteq A$ (if Π is coherent) or a reason of incoherence $\mathcal{R} \subseteq A$ (if Π is incoherent). During a debugging session the solver is first invoked by providing as input the program produced by the GRINGO-WRAPPER, that is, $\Pi = (\Delta^*_\Pi \cup \Pi_T)^G$, and $A = \mathcal{A}^{\mathcal{D}} \cup \mathcal{A}^{\mathcal{S}}$. In case an answer set is found the execution terminates and a message outlining the condition is provided to the user. Otherwise, a reason of incoherence $\mathcal{R}$ is returned by WASP. Note that, as argued in Alviano and Dodaro (2016), reasons of incoherence computed by WASP are not minimal in general. Therefore, DWASP computes a minimal reason of incoherence $\mathcal{R}^*$ by

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

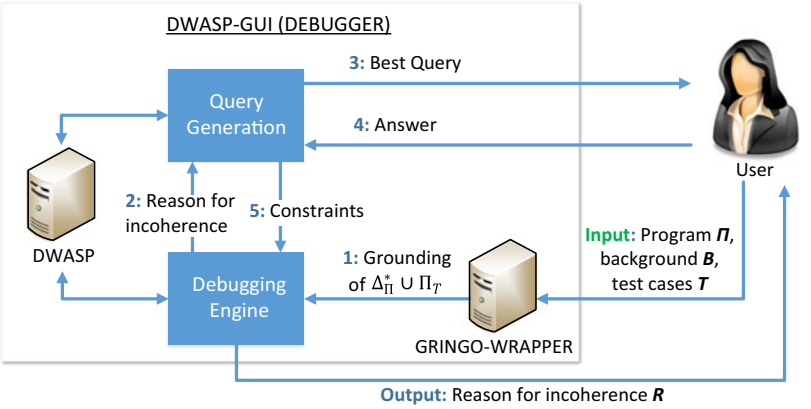


Fig. 2. Interaction of the user with the debugging system: The front-end DWASP-GUI uses GRINGO-WRAPPER and DWASP to debug the program.

using the state-of-the-art algorithm QUICKXPLAIN (Junker 2004). $\mathcal{R}^*$ is then provided to the user. Subsequently, DWASP computes one or more query atoms according to the definitions of previous section. Actually, in order to reduce the number of queries provided to the user, DWASP implements a heuristic for the computation of query atoms. Given a minimal reason of incoherence $\mathcal{R}^*$, $\mid \mathcal{R}^* \mid$ calls to the solver are performed. In particular, for each element $p \in \mathcal{R}^*$ a call is performed where $\Pi = \Delta^*_\Pi \cup \Pi_T$ and $A = \mathcal{R}^* \backslash \{p\}$. A heuristically limited number of answer sets for each call is then used to compute an estimation of the sets Q^+ and Q^- [note that the heuristic is needed to avoid to compute all possible answer sets; Dodaro *et al.* (2015)]. Then the queries are computed, and user may provide answers according to the expected solution. The answers are added in the solver input as described in previous section, and the process is repeated (computing a smaller reason of incoherence) until the bug is identified (or the user stops the debugger).

4.2 The DWASP-GUI

Architecture. The architecture of the visual debugging system is depicted in Figure 2. There are two main components: the DWASP-GUI and the debugger presented in the previous section. The DWASP-GUI implements the graphical user interface, handles input and output, and controls the invocation of the debugger. In particular, the DWASP-GUI wraps both GRINGO-WRAPPER and DWASP during the entire debugging session, so that the user can control them from a more friendly graphical environment. The components implement an interaction protocol that allows to exchange information and maintain the debugger in execution until it is interrupted by the user.

User interface. An instance of the DWASP-GUI running Example 1 is depicted in Figure 3(a). The main window is split in two parts. The panels devoted to the specification of the inputs are on the left. There, the files containing the ASP program in input are listed below the label "Workspace", and the files containing test cases are listed below the label "Test Case". Indeed, the user can specify several test case for the same program, and the interface allows to debug one case at time. Test cases are provided by the user as text files according to a simple syntax. For each atom a that is expected

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

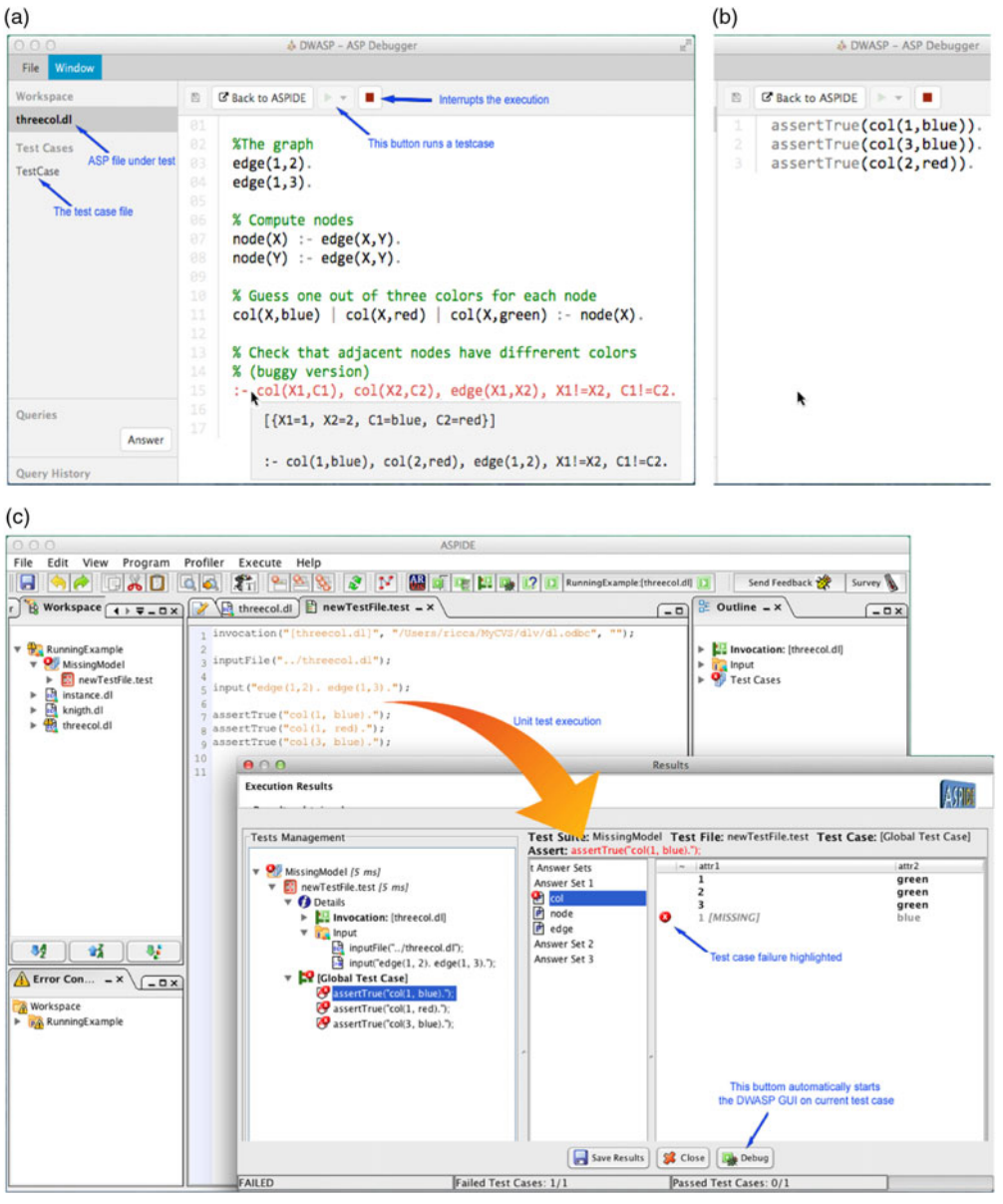


Fig. 3. Debugging and testing the 3-Colorability encoding (Example 1). (a) Main window of the DWASP-GUI. (b) A test case in DWASP-GUI. (c) Unit testing and debugging (interaction with ASPIDE).

to be true (resp. false) in the answer set, the user writes a statement *assertTrue(a)* [resp. *assertFalse(a)*]. The test case of Example 1 was encoded as depicted in Figure 3(b). On the right middle part of the window [see Figure 3(a)] there is a program editor featuring syntax highlights, where the user can edit both program and test case files. On top of the program editor is a tool-bar containing the buttons for running or concluding a debugging session on a specific test case. The button with a red square icon is used to stop a running debugger. The test case to run can be selected from a drop-down list

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

that is enabled by clicking on the run button (having a green triangle as icon). When a debugging session is started, DWASP returns a minimal reason of incoherence $\mathcal{R}$, which is interpreted by the DWASP-GUI by highlighting the corresponding non-ground rules in red. In particular, atoms of the form $\_debug(i, \dots)$ cause the highlight of the i-th rule, and when there are only atoms of the form $\_support(a(\dots))$, the rules having atoms of predicate a in the head are highlighted. The user can inspect a highlighted rule hovering over such a rule with the cursor, and the DWASP-GUI shows in a pop-up both a substitution and a ground version of the rule causing the incoherence. To have an idea of this functionality, the faulty constraints of Example 1 are shown highlighted and inspected by the user in Figure 3(a).

The identification of the faulty constraints of Example 1 is rather straightforward. To illustrate how the interface handles more complex programs, and demonstrating the query feature of DWASP, we purposely modified (introducing a simple bug in the last rule) the encoding of the *Knight Tour* used in the ASP Competitions 2011. The result obtained by running the debugger on a simple instance of that problem with the buggy encoding is depicted in Figure 4(b). In this case, DWASP identifies at first a number of rules as cause for the incoherence, but just one is guilty. At the same time DWASP computes a set of possible queries to be answered by the user. Queries are displayed in DWASP-GUI on the left panel, and the user can answer yes (resp. no) by clicking on the green (resp. red) sign, or can leave the query unanswered. Note that the user can answer several queries at once and in random order, a feature not available in the command-line interface of DWASP. In our debugging session we answer that we expect *cell(3,2)* and *reached(3,2)* to be true, and as a result DWASP is able to precisely identify the bug as depicted in Figure 4(a), and no further query can be posed to the user. Note that, on the left DWASP-GUI displays a query history where the answers given by the user can be inspected and possibly unrolled.

4.3 Integration with ASPIDE

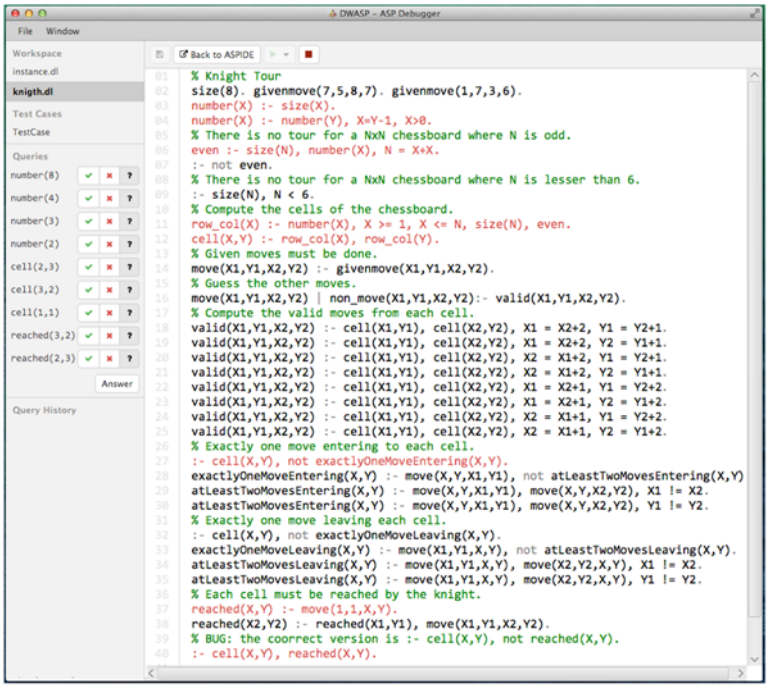
We integrated the graphical user interface DWASP-GUI inside the IDE ASPIDE (Febbraro *et al.* 2011) by developing a plug-in connector and extending some components of the user interface.

As a result, ASPIDE offers several options for invoking the DWASP-GUI. In the simplest scenario the user has to press a button in the main tool bar of ASPIDE. This button (having a bug as icon) is pointed by a blue arrow in Figure 5(a) and starts the debugger using current run configuration. Alternatively, he/she can run the DWASP-GUI on some specific files of a project. Figure 5(a) depicts also this use case, where the user (i) selects some files to debug in the project explorer (they are highlighted in blue on the left-hand side) and (ii) right-clicks the mouse to select (iii) the menu item labeled "Debug Directly" (that starts the DWASP-GUI).

Another handy option available in ASPIDE lets the user call the debugger directly from the window presenting the results of the execution of a solver. In Figure 5(b) we see that current execution terminated and no answer set has been found; thus the user can start the debugger directly from that window by clicking on the dedicated button labeled "Debug" from the toolbar in the bottom left side of the window.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

(a)



(b)

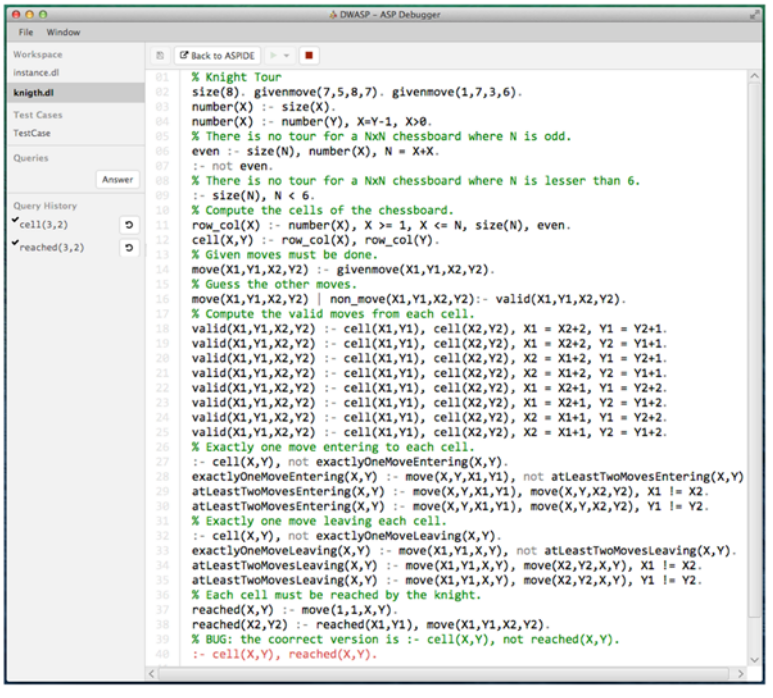


Fig. 4. Debugging the Knight Tour encoding from ASP Competition 2011. (a) Knight Tour (queries). (b) Knight Tour (identified).

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

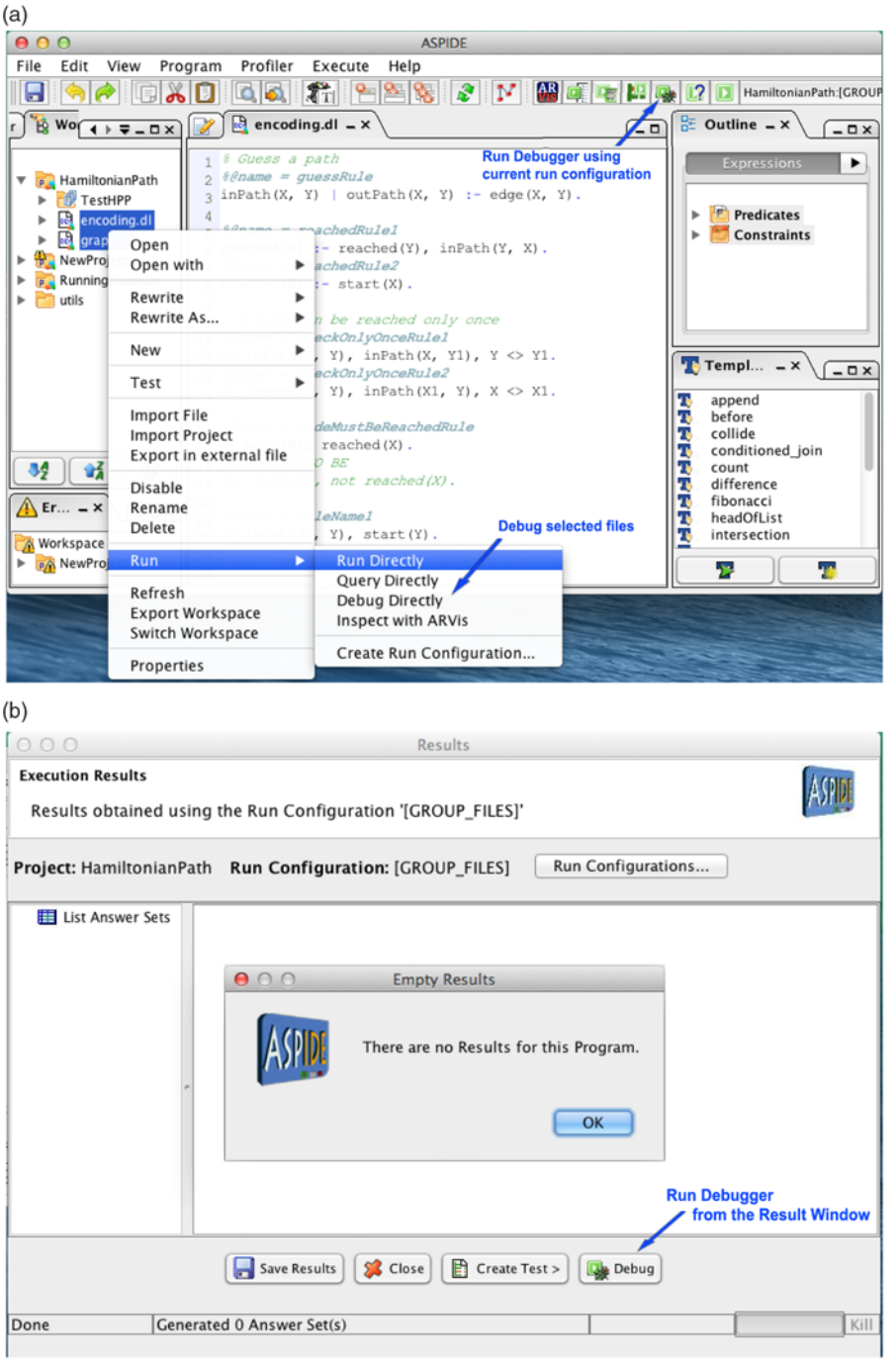


Fig. 5. Debugging the Knight Tour encoding from ASP Competition 2011. (a) Run from the main window. (b) Run from results window.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

In all the mentioned use cases the user has to do nothing for configuring DWASP-GUI, all the needed intermediate files are automatically generated by ASPIDE, and the debugger is launched automatically on the specified program.

We now use a more general use case to present the combination of the new debugger with the unit testing framework of the IDE (Febbraro *et al.* 2011). Unit testing is a white-box testing technique that requires to assess separately subparts of a source code called units to verify whether they behave as intended. ASPIDE supports a testing language that allows the developer to specify the rules composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the expected behavior of sub-programs. Test case specifications can be developed and run in ASPIDE, and the assertions are automatically verified by analyzing the output of the execution. ASPIDE provides the user with some graphic tools for simplifying the development and the inspection of results of test cases executions.[2] In the ASPIDE testing tool one could easily identify a failing test case, but there was no support for understanding the cause of failure of a test case. We solved this issue by connecting the debugger in the unit testing framework.

The workflow for testing and debugging is now illustrated by using the program and the test case presented in Example 2. In Figure 3(c), we present a screenshot of ASPIDE with a workspace that has the buggy graph colorability encoding loaded (see the file *threecol.dl* on the left panel of ASPIDE). Test cases in ASPIDE can be defined according to a rich test case specification language that was introduced in Febbraro *et al.* (2011), and that inspired the specification of test cases in DWASP. Actually, every DWASP test case is also a valid unit test case specification for ASPIDE, modulo some additional syntactic construct needed to configure the testing tool with the program file to be tested and its input. According to Example 2 we defined the test case in file *newTestFile.test*, and its specification is shown in the central editor window of ASPIDE of Figure 3(c). When test cases are executed, a new window is opened [small window in Figure 3(c)], where the result of the execution is shown. Failing test cases are highlighted in red. The user can start debugging of one of the failing test case by just clicking on the *Debug* button, and the DWASP-GUI window of Figure 3(a) is displayed. Note that, once more the user has to do nothing for configuring DWASP-GUI, all the needed files are automatically generated by ASPIDE. Finally, the *Back to ASPIDE* button allows to see the faulty rule highlighted also in ASPIDE.

5 Performance analysis

We have assessed the performance of our implementation by comparing it with the debugger OUROBOROS (Oetsch *et al.* 2010; Polleres *et al.* 2013), which is the only maintained solution able to cope with non-ground programs. In particular, we have employed the same ASP encodings and instances taken from ASP competitions that have been used in Polleres *et al.* (2013) for analyzing the performance of a debugger. The benchmark considered are Graph Colouring, Hanoi Tower, Knights Tour, and Partner Units.

[2] A complete description of the ASPIDE framework for unit testing is out of the scope of this paper, for more details we refer the reader to Febbraro *et al.* (2011).

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Table 1. *Comparison of grounding programs produced by* GRINGO-WRAPPER *and* OUROBOROS

			GRINGO		GRINGO-WRAPPER		OUROBOROS	
Benchmark	Instance	#n-ground	#ground	Time (s)	#ground	Time (s)	#ground	Time (s)
Graph Col.	1-125	1672	6145	0.22	8031	0.63	19,020	0.95
Graph Col.	11-130	1757	6455	0.21	8416	0.68	19,845	1.10
Graph Col.	21-135	1986	7269	0.24	9305	0.73	21,174	1.04
Graph Col.	30-135	1794	6597	0.25	8633	0.64	20,502	1.02
Graph Col.	31-140	2039	7467	0.22	9578	0.67	21,887	1.03
Graph Col.	40-140	2219	8097	0.32	10,208	0.68	22,517	1.03
Graph Col.	41-145	2262	8260	0.25	10,446	0.68	23,195	1.04
Graph Col.	51-120	2405	8773	0.36	11,034	0.76	24,223	1.05
Hanoi	09-28	104	31,748	0.40	94,166	1.61	1,739,800	8.09
Hanoi	11-30	106	34,056	0.33	100,942	1.58	1,864,222	9.50
Hanoi	15-34	110	38,672	0.38	114,524	2.11	2,112,986	9.43
Hanoi	16-40	100	27,137	0.35	80,615	1.40	1,491,281	7.04
Hanoi	22-60	102	28,311	0.29	84,644	1.43	1,678,483	7.80
Hanoi	38-80	106	34,044	0.23	100,942	1.68	1,864,250	8.53
Hanoi	41-100	104	31,738	0.39	94,166	1.52	1,739,830	13.24
Hanoi	47-120	99	25,968	0.19	77,227	1.49	1,429,695	6.90
Knights Tour	01-08	21	1384	0.34	3413	1.14	12,985,716	59.44
Knights Tour	03-12	22	3356	0.13	8652	0.60	>72,244,034	>300
Knights Tour	05-16	21	6192	0.16	16,285	0.64	>69,494,641	>300
Knights Tour	06-20	21	9892	0.16	26,321	0.88	>62,785,993	>300
Knights Tour	07-30	21	22,922	0.40	61,911	1.13	>59,166,564	>300
Knights Tour	08-40	21	41,352	0.44	112,501	1.27	>54,944,042	>300
Knights Tour	09-46	21	55,002	0.53	150,055	1.58	>56,443,633	>300
Knights Tour	10-50	22	65,182	0.86	178,094	2.15	>62,402,315	>300
Partner Units	176-24	68	12,563	0.22	14,218	1.03	102,023	1.47
Partner Units	23-30	117	39,231	0.29	42,106	1.20	276,645	2.11
Partner Units	29-40	108	59,979	0.34	64,413	1.67	629,639	3.35
Partner Units	207-58	136	158,564	0.61	168,289	3.07	2,726,182	11.94
Partner Units	204-67	141	218,808	0.78	231,083	5.30	4,280,282	17.79
Partner Units	175-75	290	682,015	2.10	699,472	16.03	8,604,415	40.60
Partner Units	52-100	254	952,363	2.68	979,603	16.61	20,125,857	90.10
Partner Units	115-100	254	952,369	2.86	979,759	16.07	20,317,011	94.26

For grounding we use GRINGO (v4.4.0) in both methods. The comparison is done by measuring grounding size and running time of GRINGO for both debugging tools.

Results are reported in Table 1, where the first two columns represent the benchmarks and the instances considered, respectively. The third column is the number of rules of the non-ground program, while #ground and time(s) are the size of the ground program and the execution time of GRINGO, respectively. In our approach the increase in grounding size is due to the fact that the GRINGO-WRAPPER disables the optimizations performed by GRINGO, whereas in OUROBOROS the grounding of an ASP program modeling debugging is required. Considering the instances that were groundable with GRINGO within 5 min by our Intel Core i7-3667U machine with 8 GB of RAM, we report that in our approach

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

the size of the instantiation of the debugging program is from 1.5 to 3 times the size of grounding the original program, whereas the debugging program of OUROBOROS generates groundings that are from 50 times up to 9382 times larger than the original program. Note that, the performance of our approach is only limited by the performance of the underlying solver, whereas in the case of OUROBOROS the limit is in the grounding of the debugging program, which may not be feasible.

6 Usability test

In order to assess usability of the interface and degree of appreciation for our debugger tool we have set up a usability experiment. The test has been conducted during a regular class of the course on ASP given by Prof. Nicola Leone for Bachelor Degree students in Computer Science at University of Calabria. The assessment was executed on January 19, 2017, during a regular practice class at the end of the course. The tool was not explained in a previous lecture and the experiment was not announced in advance to ensure that: (i) users (i.e. students) never used the tool before, and (ii) represent a sample of a distribution including both sufficiently skilled and less skilled ASP programmers; (iii) do not include only those that are interested in tools or have specific bias on using programming environments.

Test setup. We prepared a test in which students were asked to find a bug on three ASP encodings. We selected for this purpose tree well-known problems, and modified the encodings available from the third ASP competition (Calimeri *et al.* 2014) website for the following problems: 3-Colorability, Hamiltonian Path, and Stable Marriage. The first encoding is a classical example, which was familiar to the majority of users since it was presented during a lecture few months before to explain the guess and check methodology, and comprises only two (non-ground) rules. The encodings for the second and third problem were completely new to the audience, and they are much more complex featuring six rules each. In particular, the encoding of Stable Marriage is the least intuitive one and thus it was expected to be the hardest to fix.

We modified one constraint per encoding so that some expected answer sets were missing on a given test case. All the encodings use basic features of the language, that is, disjunctive normal logic programs as described in the founding paper by Gelfond and Lifschitz (1991) as well as in ASPCore 2.0 syntax (Calimeri *et al.* 2016). This choice ensures that the encodings and causes of faults in every test are comprehensible to the audience and no knowledge of advanced language constructs – not covered by the lectures – is required. The students were provided with complete textual descriptions of the problems, including an explanation of the signatures and meaning of input and output predicates, one buggy encoding, and one test instance per problem. Students were working on own notebooks where ASPIDE with debugger was pre-installed and launched with a pre-loaded workspace containing one project per problem with all required files: problem description, encoding, and sample instances.

The test started after providing the users with: (i) a description of the task to accomplish, (ii) some minimal instructions on how to start the debugger and operate on the main buttons of the interface using a different example from the ones used in the test, and (iii) an anonymous questionnaire with a time annotating and debugger usage sheets

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

for each problem. In these sheets the students had to give notes on elapsed time and degree to which the debugger helped to find a bug. The questionnaire, to fill at the end of the experience, contained the following questions:

- Do you find the debugger easy to use?
- Is bug detection faster using the debugger?
- Will you consider using it next time?
- How do you judge the user interface usability?

The student could answer one of *Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly agree* for the first three questions and one of *Poor, Fair, Average, Good, Excellent* for the latter.

Collection of results and hypothesis testing. We collected the results provided by 26 students on three usage tests (one per problem) of 30 min each. These number are in line with the Jakob Nielsen recommendation (Nielsen and Landauer 1993) for finding serious usability problems in user interfaces.[3] Moreover, to test the validity of our conclusions we applied the Kolmogorov–Smirnov (K–S) test on the results, that refused the null hypothesis with an accuracy $\geq$95%.[4]

Debugger applicability for complex problems. One of our test goals was to determine the impact of bug fixing complexity on the applicability of the debugger. To verify that a problem is more difficult to solve than another we measured average bug fixing times on "fixed" cases as well as the number of cases in which a bug was identified. 3-Colorability – the easiest problem – was solved by all students but one in 6.4 min on average, and only 38.5% declared the debugger was actually used for finding the bug. Hamiltonian Path required 9.1 min on average to be fixed, of which 91% declared the debugger was used, and only one student failed the test. Stable marriage was fixed in 8.2 min on average, and 100% of the students declared the debugger was used, and two students failed the test. For the sake of completeness, we observed that the student failing in 3-Colorability, failed also on Stable Marriage, and could solve Hamiltonian path in 25 min (the maximum, and clearly an outlier), thus we believe this was just a non-proficient student with limited understanding of the language. Thus, from this findings we conclude that the debugger was used more (and, thus empirically it was more useful) as the complexity of finding the bug increases.

Probing the opinion of the users. Results of the questionnaire are summarized in Figure 6. All students agreed that the debugger is easy to use – 77% of students answered "Agree" and the 23% "Strongly agree" to the first question as shown in Figure 6(a). Concerning the second question [see Figure 6(b)], 64% conclude that using the debugger accelerates the bug fixing process (of which 8% strongly agrees), 8% is neutral, and 30% disagrees. Interestingly, the last group includes all those students that failed at least one test as well as some of those that did not use the debugger for some test. We interpret this

[3] The Jakob Nielsen claim roughly says that few testers (no more than five users) and running as many small tests as you can afford is enough to identify a serious usability problem (Nielsen and Landauer 1993).

[4] The K–S test is one of the most useful and general non-parametric tests that we used because it is more powerful than other methods (e.g. χ^2 tests) when the size of the sample is below 50 elements, and some events (possible answers) have low frequency.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

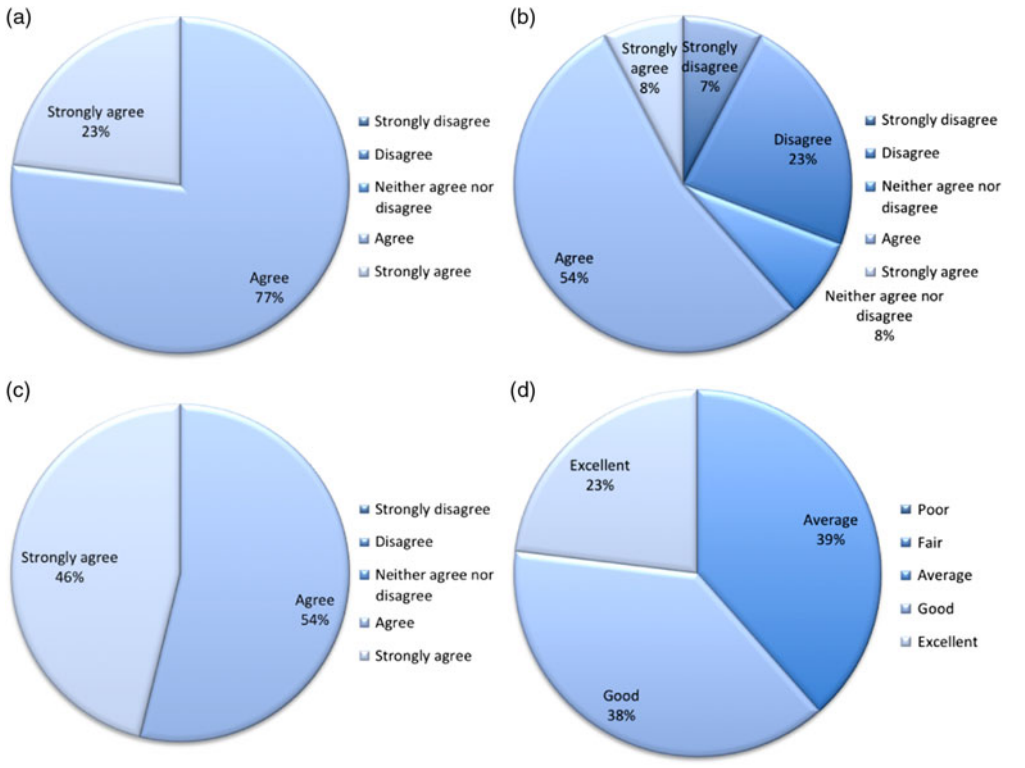


Fig. 6. Results of the usability testing. (a) Do you find the debugger easy to use? (b) Is bug detection faster using the debugger? (c) Will you consider using it next time? (d) How do you judge the user interface usability?

result as follows: since the debugger was not needed to solve the easiest test, one cannot agree in general that it always makes bug fixing faster. This explanation agrees with the message that comes from results to the third question presented in Figure 6(c). In this case all students would consider using the debugger again – actually 46% strongly agree. This further outlines that also the critical users found it better to have our tool at their disposal. Finally, with the last question we asked whether they were satisfied with the user interface. The results are positive [see Figure 6(d)]: no student found the interface insufficient, 38% claims it is a good interface, and 23% finds it excellent.

We can conclude that our debugger was considered easy to use, and effective when the difficulty of bug-fixing is high; moreover, no serious usability problem was revealed, and the user interface was perceived to be largely acceptable.

7 Related work

There are multiple approaches to ASP debugging suggested in the literature including algorithmic (Brain and De Vos 2005; Syrjänen 2006), stepping-based (Oetsch *et al.* 2011), and meta-programming (Brain *et al.* 2007; Gebser *et al.* 2008; Oetsch *et al.* 2010; Polleres *et al.* 2013; Shchekotykhin 2015) methods. Among the algorithmic approaches IDEAS (Brain and De Vos 2005) aims at explaining: (a) why a set of atoms S is in an answer set M, and (b) why S is not in any answer set. IDEAS allows a

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

programmer: (1) to query for an explanation of an observed fault, (2) to analyze the obtained results, and (3) reformulate the query to make it more precise. In our approach refinements are found automatically once the user provides additional knowledge on an expected answer set, thus, making the steps (2) and (3) obsolete.

Meta-programming debuggers use a program over a meta-language – a kind of ASP solver simulation – to manipulate a program over an object language – the faulty program. Each answer set of a meta-program comprises a *diagnosis*, which is a set of meta-atoms describing the cause why some interpretation of the faulty program is not its answer set. The SPOCK (Gebser *et al.* 2008) and OUROBOROS (Oetsch *et al.* 2010; Polleres *et al.* 2013) debuggers enable the identification of faults connected with over-constraint problems and unfounded sets. Both approaches represent the input program in a reified form allowing application of a debugging meta-program. In case of SPOCK the debugging can be applied only to grounded programs, whereas OUROBOROS can tackle non-grounded programs as well. Our approach does not fall in the meta-programming classification because it does not need any reification, nor a specific debugging program that manipulates the reified input program. These design choices are the main reason why meta-programming are affected by the grounding blowup (the grounding of the meta-program could be huge) (Polleres *et al.* 2013). Thus, the ground debugging program has to comprise all atoms explaining all possible faults in an input faulty program, which is not the case in our approach. Moreover, our approach generalizes the query-based method built on top of SPOCK (Shchekotykhin 2015) by enabling its application to non-ground programs. We also observe that our approach works in a radically different way with respect to meta-programming ones, since we just add a marker to each rule and compute (and minimize) reasons of incoherence. Another difference is that OUROBOROS, in case the bug is caused by an unfounded loop, is able to provide a loop comprising the atom. This information is missing in our approach, which just treats unfounded loops as missing support.

The approach of SMDEBUG (Syrjänen 2006) addresses debugging of incoherent non-disjunctive ASP programs by adaption of Reiter's model-based diagnosis. Similarly to our approach the debugger focuses on analyzing contradictions, but cannot detect problems arising due to some atom missing support (since only odd loops are considered to be errors).

There are other approaches enabling faults localization in ASP, but not directly comparable with DWASP, including Consistency-Restoring Prolog (Balduccini and Gelfond 2003), translation of ASP programs to natural language (Mikitiuk *et al.* 2007), visualization of justifications for an answer set (Pontelli *et al.* 2009) as well as stepping through an ASP program (Oetsch *et al.* 2011). In Li *et al.* (2015), the authors present a debugging technique for normal ASP programs that is based on inductive logic programming (ILP) and test cases. The idea is to allow the programmer to specify test cases modeling features that are expected to appear in some solution and those that should not. These are used to revise the original program semi-automatically so that it satisfies the stated properties. This approach offers the possibility to learn rules (and modifications of rules), whereas DWASP focus only on identifying the buggy rules of a given program. Combining these approaches with ideas implemented in DWASP is part of our future work.

In Schulz *et al.* (2015) and Schulz and Toni (2016) bugs are studied in terms of a set of culprits (atoms) using semantics which are weaker than the answer set semantics. A technique for explaining the set of culprits in terms of derivations is also provided.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Approaches explaining bugs with the truth of a set of atoms are, in a sense, complementary to our approach (we identify the rules involved in a conflict).

In Dasseville and Janssens (2015) the web-based programming environment for the IDP system is presented that also features a debugging approach based on the computation of a reason of incoherence. This debugger does not feature a question-answering schema that is fundamental for reducing the set of buggy rules. Moreover, we are not aware of any IDE for ASP that provides a tight combination of debugging and unit testing environments as the one presented in this paper.

8 Conclusion

ASP features an intuitive syntax and a well-known semantics, nonetheless the process of finding bugs in logic programs can be non-trivial and is often a tedious task. For this reason, valid ASP debuggers have emerged during the recent years. The most prominent approaches, using ASP itself to compute explanations, are however affected by two main issues somehow limiting their applicability some in practical cases: (i) the grounding blowup, which may make impossible to compute the causes of a bug; and (ii) the overwhelming number of produced explanations, which might be impossible to be browsed by users.

In this paper we propose a novel debugging approach for non-ground ASP programs that is not affected by both the above issues. Indeed, it points the user directly to a set of rules involved in the bug, and – importantly – allows to refine that set interactively by asking the user-specific questions on an expected answer set, until the bug can be easily identified.

The new approach has been implemented in the DWASP Debugger, which was obtained by properly combining the grounder GRINGO with an extended version of the ASP solver WASP. An empirical analysis shows that the new debugger is not affected by the grounding blowup and can handle instances that are pragmatically out of reach for state-of-the-art meta-programming-based debuggers.

The DWASP Debugger has been complemented by a user-friendly graphical interface, called DWASP-GUI. The graphical interface improves the user-experience of debugging ASP programs, as demonstrated by running a usability test on a class of students attending a university course on ASP. Indeed, besides the usual advantages provided by visual tools, the DWASP-GUI simplifies two tasks that are not easy to carry out in the command line interface, namely: the definition of test cases and the interactive query answering. The query-answering feature is made much more user-friendly, since the user can simply select answers by clicking on dedicated buttons, and several possible answers are presented to the user in a convenient list. Problematic rules are outlined immediately in the text editor so the user is pointed immediately from the interface to sources of bugs. DWASP-GUI has also been integrated in ASPIDE, which was missing a complete debugger interface supporting non-ground ASP programs. The integration includes specific support for creating failing test cases to debug directly from the unit test framework provided by ASPIDE supporting test-driven development. The rapid identification of the cause of a failig test case is fundamental for test-driven development (Fraser *et al.* 2003). With our extension ASPIDE turns into a more complete IDE by offering improved debugging support and a more effective test-driven development environment.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Concerning future works, one possibility would be to study a possible integration of our approach with existing ones. Moreover, an interesting work would be to investigate if our debugging approach can be generalized also in the case when an extra, incorrect, answer set is provided. We also plan to extend the tool in order to better handle some specific bugs related to missing support, in particular those due to the so-called *unfounded sets*. *Availability.* The DWASP-GUI can be obtained from https://github.com/gaste/dwasp-gui, and ASPIDE from http://www.mat.unical.it/ricca/aspide, the plugin connector installation starts the first time the debugger is launched.

References

Abseher, M., Gebser, M., Musliu, N., Schaub, T. and Woltran, S. 2016. Shift design with answer set programming. *Fundamenta Informaticae 147,* 1, 1–25.

Alviano, M. and Dodaro, C. 2016. Anytime answer set optimization via unsatisfiable core shrinking. *TPLP 16,* 5–6, 533–551.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *LPNMR*. Lecture Notes in Computer Science, vol. 9345. Springer, Berlin, 40–54.

Alviano, M., Dodaro, C. and Maratea, M. 2017. An advanced answer set programming encoding for nurse scheduling. In *AI*IA*. Lecture Notes in Computer Science, vol. 10640. Springer, Berlin, 468–482.

Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A. and Thorstensen, E. 2011. Optimization methods for the partner units problem. In *CPAIOR*. Lecture Notes in Computer Science, vol. 6697. Springer, Berlin, 4–19.

Balduccini, M. and Gelfond, M. 2003. Logic programs with consistency-restoring rules. In *AAAI Spring Symposium*, AAAI, 9–18.

Balduccini, M., Gelfond, M., Watson, R. and Nogueira, M. 2001. The USA-advisor: A case study in answer set planning. In *LPNMR*. Lecture Notes in Computer Science, vol. 2173. Springer, Berlin, 439–442.

Baral, C. 2010. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

Brain, M. and De Vos, M. 2005. Debugging logic programs under the answer set semantics. In *Answer Set Programming*. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org.

Brain, M., Gebser, M., Schaub, T., Tompits, H. and Woltran, S. 2007. "That is Illogical Captain!" – The debugging support tool `spock` for answer-set programs: System description. In *SEA*, 71–85.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Busoniu, P., Oetsch, J., Pührer, J., Skocovsky, P. and Tompits, H. 2013. SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support. *TPLP 13,* 4–5, 657–673.

Calimeri, F., Gebser, M., Maratea, M. and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence 231*, 151–181.

Calimeri, F., Ianni, G. and Ricca, F. 2014. The third open answer set programming competition. *TPLP 14,* 1, 117–135.

Dasseville, I. and Janssens, G. 2015. A web-based IDE for IDP. *CoRR abs/1511.00920*.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F. and Schekotihin, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP 16,* 5–6, 653–669.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Dodaro, C., Gasteiger, P., Musitsch, B., Ricca, F. and Shchekotykhin, K. M. 2015. Interactive debugging of non-ground ASP programs. In *LPNMR*. Lecture Notes in Computer Science, vol. 9345. Springer, Berlin, 279–293.

Dodaro, C., Leone, N., Nardi, B. and Ricca, F. 2015. Allotment problem in travel industry: A solution based on ASP. In *RR*. Lecture Notes in Computer Science, vol. 9209. Springer, Berlin, 77–92.

Eiter, T., Gottlob, G. and Mannila, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems 22,* 3, 364–418.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

Erdem, E. and Öztok, U. 2015. Generating explanations for biomedical queries. *TPLP 15,* 1, 35–78.

Febbraro, O., Leone, N., Reale, K. and Ricca, F. 2011. Unit testing in ASPIDE. In *INAP/WLP*. Lecture Notes in Computer Science, vol. 7773. Springer, Berlin, 345–364.

Febbraro, O., Reale, K. and Ricca, F. 2011. ASPIDE: Integrated development environment for answer set programming. In *LPNMR*. Lecture Notes in Computer Science, vol. 6645. Springer, Berlin, 317–330.

Fraser, S., Beck, K. L., Caputo, B., Mackinnon, T., Newkirk, J. and Poole, C. 2003. Test driven development (TDD). In *XP*. Lecture Notes in Computer Science, vol. 2675. Springer, Berlin, 459–462.

Gavanelli, M., Nonato, M. and Peano, A. 2015. An ASP approach for the valves positioning optimization in a water distribution system. *Journal of Logic and Computation 25,* 6, 1351–1369.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, San Rafael.

Gebser, M., Kaminski, R., König, A. and Schaub, T. 2011. Advances in *gringo* series 3. In *LPNMR*. Lecture Notes in Computer Science, vol. 6645. Springer, Berlin, 345–351.

Gebser, M., Maratea, M. and Ricca, F. 2015. The design of the sixth answer set programming competition – Report. In *LPNMR*. Lecture Notes in Computer Science, vol. 9345. Springer, Berlin, 531–544.

Gebser, M., Maratea, M. and Ricca, F. 2016. What's hot in the answer set programming competition. In *AAAI*. AAAI Press, Palo Alto, CA, USA, 4327–4329.

Gebser, M., Pührer, J., Schaub, T. and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *AAAI*. AAAI Press, Palo Alto, PA, USA, 448–453.

Gebser, M., Schaub, T., Thiele, S. and Veber, P. 2011. Detecting inconsistencies in large biological networks with answer set programming. *TPLP 11,* 2–3, 323–360.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Grasso, G., Iiritano, S., Leone, N. and Ricca, F. 2009. Some DLV applications for knowledge management. In *LPNMR*. Lecture Notes in Computer Science, vol. 5753. Springer, Berlin, 591–597.

Grasso, G., Leone, N., Manna, M. and Ricca, F. 2011. ASP at work: Spin-off and applications of the DLV system. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 6565. Springer, Berlin, 432–451.

Junker, U. 2004. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI*. AAAI Press, Palo Alto, CA, 167–172.

Koponen, L., Oikarinen, E., Janhunen, T. and Säilä, L. 2015. Optimizing phylogenetic supertrees using answer set programming. *TPLP 15,* 4–5, 604–619.

https://doi.org/10.1017/S1471068418000492 Published online by Cambridge University Press

Li, T., De Vos, M., Padget, J., Satoh, K. and Balke, T. 2015. Debugging ASP using ILP. In *ICLP (Technical Communications)*. CEUR Workshop Proceedings, vol. 1433. CEUR-WS.org.

Lierler, Y., Maratea, M. and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine 37,* 3, 45–52.

Mikitiuk, A., Moseley, E. and Truszczynski, M. 2007. Towards debugging of answer-set programs in the language PSpb. In *IC-AI*. CSREA Press, Athens, GA, USA, 635–640.

Nielsen, J. and Landauer, T. K. 1993. A mathematical model of the finding of usability problems. In *INTERCHI*. ACM, New York, NY, USA, 206–213.

Oetsch, J., Pührer, J. and Tompits, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *TPLP 10,* 4–6, 513–529.

Oetsch, J., Pührer, J. and Tompits, H. 2011. Stepping through an answer-set program. In *LPNMR*. Lecture Notes in Computer Science, vol. 6645. Springer, Berlin, 134–147.

Polleres, A., Frühstück, M., Schenner, G. and Friedrich, G. 2013. Debugging non-ground ASP programs with choice rules, cardinality and weight constraints. In *LPNMR*. Lecture Notes in Computer Science, vol. 8148. Springer, Berlin, 452–464.

Pontelli, E., Son, T. C. and El-Khatib, O. 2009. Justifications for logic programs under answer set semantics. *TPLP 9,* 1, 1–56.

Schulz, C., Satoh, K. and Toni, F. 2015. Characterising and explaining inconsistency in logic programs. In *LPNMR*. Lecture Notes in Computer Science, vol. 9345. Springer, Berlin, 467–479.

Schulz, C. and Toni, F. 2016. Justifying answer sets using argumentation. *TPLP 16,* 1, 59–110.

Shchekotykhin, K. M. 2015. Interactive query-based debugging of ASP programs. In *AAAI*. AAAI Press, Palo Alto, CA, USA, 1597–1603.

Silva, J. P. M. and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2, 181–234.

Syrjänen, T. 2006. Debugging inconsistent Answer Set Programs. In *NMR*, 77–84.