Monitoring and Visualizing Answer Set Solving

Arne König and Torsten Schaub
Institut für Informatik, Universität Potsdam

submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

A distinguishing feature of Answer Set Programming (ASP) is its *declarativity*, decoupling problem representations from problem solving algorithms. However, this strict separation should not lead to viewing the solving process as a magic black box refusing any insights into how the problem is solved. We address this issue and propose a two-fold approach for enhancing the transparency of the solving process. At first, we provide a flexible data logger protocoling relevant events occurring during ASP solving. This module is system-specific and realized with the ASP solver *clasp*. The recorded information can subsequently be used in various ways by back-ends of choice. With such data at hand, we then furnish a visualization back-end offering various views on the underlying problem structure as well as the solving process over time. Together both tools allow us to re-connect the solving process with the original problem specification and thus to reveal how the original problem is actually solved.

1 Introduction

Answer Set Programming (ASP; (Baral 2003)) is an approach to declarative problem solving that combines a rich yet simple modeling language with high-performance solving capacities. A distinguishing feature of ASP lies in its high declarativity, strictly separating a problem's representation from the algorithms used for solving it. As a matter of fact, this brings about new challenges given that traditional software engineering techniques relying on the connection between program specification and execution are inapplicable. A prominent example is the failure of procedural techniques like tracing. This separation is even enlarged during ASP's solving process transforming first-order programs into a propositional format. In this way, the combination of a few first-order rules with a large set of facts may lead to a vast set of propositional rules. Furthermore, modern conflict-driven ASP solvers turn the resulting set of rules into an even larger set of Boolean constraints being subject to solving.

However, this strict separation should not lead to viewing the solving process as a magic black box refusing any insights into how the problem is solved. We address this by greatly enhancing the transparency of the solving process. We accomplish this by a two-fold approach. At first, we provide a flexible data logger protocoling relevant events occurring during ASP solving. The recorded information can subsequently be used in various ways by back-ends of choice. With such data at hand, we then furnish a visualization back-end offering various views on the underlying problem structure as well as the solving process over time. Together both tools allow us to re-connect the solving process with the original problem specification and thus to reveal how the original problem is actually solved. We have implemented our data logger as an extension to the conflict-driven constraint learning ASP solver *clasp* (Gebser et al. 2012). The resulting system, called *clavis*, is easily configurable and allows for monitoring all (single-threaded) configurations of *clasp*. The issuing data is provided as an event series reflecting the solving process, and stored as a queryable database. This event series serves as input to our solver-independent visualization tool *insight*, providing various structural and temporal perspectives on the solving process. The structural views rely on graphs for projecting different forms of variable dependencies, like occurrences in the same program or conflict constraint. To provide enriched node information, *insight* exploits the ASP solver's symbol table to link solver variables to ground atoms of the original problem specification. This is accompanied with a simple query language that allows us to restrict the projection to variables satisfying a given query. While the structural perspectives come with alternative variable-specific values aggregated over the solving process, the different temporal views aim at capturing the dynamics of the solving process. Hence, they focus on indicative algorithmic figures, like the development of the conflict or decision level over time. Such developments are provided as two-dimensional plots. Within these plots particular event segments can be selected to induce in turn structural perspectives restricted to the aggregated values of the period in focus.

Our data logger *clavis* and our visualizer *insight* are both freely available at (clavis).

2 Background

We assume some familiarity with ASP, its semantics as well as its basic language constructs. A comprehensive treatment of ASP can be found in (Baral 2003).

Once a problem is modeled as a (first-order) logic program, ASP solving proceeds in two steps. First, a grounder generates a finite propositional representation of the input program. After that, a solver computes the stable models of the propositional program. The resulting stable models represent the solutions to the original problem.

For computing the stable models of a logic program by means of modern Boolean constraint technology, the problem must be expressed in terms of Boolean constraints. For this, we rely on *nogoods* (Dechter 2003) representing invalid partial assignments. A *solution* for a set of nogoods is then a total (Boolean) assignment excluding all nogoods. While clauses can be directly mapped into nogoods, logic programs are subject to a more complex translation, often involving the introduction of auxiliary (propositional) variables. For instance, by abbreviating elementary Boolean assignments $x \mapsto \boldsymbol{T}$ and $x \mapsto \boldsymbol{F}$ by signed literals of form $\boldsymbol{T}x$ and $\boldsymbol{F}x$, respectively, an atom a defined by two rules '$a \leftarrow b, {\sim}c$' and '$a \leftarrow d$' gives rise to three nogoods: $\{\boldsymbol{T}a, \boldsymbol{F}x_{\{b,{\sim}c\}}, \boldsymbol{F}x_{\{d\}}\}$, $\{\boldsymbol{F}a, \boldsymbol{T}x_{\{b,{\sim}c\}}\}$, and $\{\boldsymbol{F}a, \boldsymbol{T}x_{\{d\}}\}$, where $x_{\{b,{\sim}c\}}$ and $x_{\{d\}}$ are auxiliary variables for the bodies of the two previous rules. Similarly, the body $\{b, {\sim}c\}$ leads to nogoods $\{\boldsymbol{F}x_{\{b,{\sim}c\}}, \boldsymbol{T}b, \boldsymbol{F}c\}$, $\{\boldsymbol{T}x_{\{b,{\sim}c\}}, \boldsymbol{F}b\}$, and $\{\boldsymbol{T}x_{\{b,{\sim}c\}}, \boldsymbol{T}c\}$. The last nogood precludes solutions assigning true to both variables $x_{\{b,{\sim}c\}}$ and c. See (Gebser et al. 2012) for full details.

Once a logic program is translated, we can take advantage of *Conflict-Driven Constraint Learning* (CDCL; (Marques-Silva and Sakallah 1999; Zhang et al. 2001)) for computing the solutions of the obtained set of nogoods. The basic algorithm is outlined in Fig. 1. The CDCL algorithm first extends a given (partial) *assignment* via deterministic (unit) propagation. Importantly, every derived literal is "forced" by some *nogood* (seen as a set of signed literals that must not jointly be assigned), which would be violated if the literal's complement were assigned. Although propagation aims at forgoing nogood violations, assigning a literal forced by one nogood may lead to the violation of another nogood; this situation is called *conflict*. If the conflict can be resolved (the violated nogood contains backtrackable literals), it is analyzed to identify a conflict constraint. The latter represents a "hidden" conflict reason that is recorded and guides backjumping to an earlier stage such that the complement of some formerly assigned literal is forced by the

```
loop
    propagate                                          // compute deterministic consequences
    if no conflict then
        if all variables assigned then return variable assignment
        else decide                                    // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            analyze                                    // analyze conflict and add a conflict constraint
            backjump                                   // undo assignments until conflict constraint is unit
```

Fig. 1. Basic algorithm for conflict-driven Boolean constraint learning (CDCL)

conflict constraint, thus triggering propagation. Only when propagation finishes without conflict, a (heuristically chosen) literal can be assigned at a new *decision level*, provided that the assignment at hand is partial, while a *solution* (total assignment not violating any nogood) has been found otherwise. The eventual termination of CDCL is guaranteed, by either returning a solution or encountering an unresolvable conflict (independent of unforced decision literals). In practice, CDCL employs further operations promoting the search process. One such operation consists in occasionally restarting the search process in order to escape from unfruitful search spaces while keeping gathered information. This could be added to Fig. 1 by replacing "*decide*" by "*decide* **or** *restart*". Another crucial operation is nogood *deletion* given that an exponential number of nogoods is learnable. Conceptually, this is commonly performed after *propagate* in Fig. 1.

3 Data logging and visualization

As sketched in the introductory section, our overall approach is two-fold, consisting of an online data logging phase along with an offline (visual) analysis phase.

Online data logging. The data logger protocols events relevant to CDCL-based solving during an actual run of an ASP solver.

To begin with, it records all deterministic consequences derived in *propagate* and non-deterministic assignments done by *decide* along with the respective decision levels. These events can already be used to extract various interesting figures, like how often did a variable change its truth value, how often is a variable implied or chosen, how many propagations follow a particular choice, at which decision level was a variable implied or chosen, etc.

An operation crucial to CDCL is conflict analysis, accomplished by *analyze*. Central to this is a resolution derivation, resolving conflict constraints with constraints used in unit propagation. All such derivations are recorded during data logging. This also provides the resulting conflict constraint, which is learned by CDCL. Moreover this allows us to track how often variables (jointly) occur in conflict resolutions and the resulting conflict constraints. In fact, conflict information is central to CDCL because it is used for various heuristics, as for instance in *decide* and constraint *deletion*. Notably, the generality of our approach allows for monitoring and comparing different decision heuristics in a uniform setting. For example, heuristics like *berkmin* (Goldberg and Novikov 2002) or *vsids* (Moskewicz et al. 2001) rely on different ways of scoring variables according to their conflict involvement.

Similarly, the data logger records all *backjump* and *restart* operations along with the respective decision levels. This involves the skipped decision levels as well as data on backtracked assignments. Similar to the above, this can moreover be combined with data on the subsequent *propagate* operation for determining the resulting deterministic consequences.

Finally, the data logger also keeps track of constraint *deletion* and incorporates static data, like program constraints, resulting stable models, and symbol tables, one mapping original atoms to grounder identifiers and another mapping original and auxiliary atoms to solver identifiers. Although the logger's implementation is necessarily solver-specific, our design was driven by the desire to extract only information pertinent to the CDCL-based solving process and to exclude any implementation-specific data. As a result, our data logger produces a database comprising an event series reflecting the ASP solving process along with some static data. The obtained information can then be used in various ways by different back-ends.

Offline visual analysis. Our primary back-end aims at visualizing the gathered information to provide insights into the ASP solving process. To this end, we (currently) focus on structural and temporal perspectives.

Structural aspects. For capturing structural aspects, we concentrate on *interaction graphs*[1] being undirected graphs providing a uniform abstraction of often richer yet dedicated structures. While the set of vertices is fixed to (a subset of) the solver's propositional variables, the set of edges varies in view of the type of interaction to be displayed. In its original definition, the interaction graph connects two variables whenever they are contained in the same program clause. Extending this concept, we provide graphs displaying interactions indicating containment in the same program constraint (cf. Fig. 2), learned nogood, resolution derivation, or conflict nogood. Unlike these, the choice tree graph contains only variables that have been non-deterministically assigned (in *decide*); it links two variables whenever one was decided after the other (without intermediate *backjump* or *restart*).

Given that interaction graphs have no predefined layout, we follow (Sinz and Dieringer 2005) in using a *force-directed graph drawing* algorithm (Hachul and Jünger 2004) for rendering. These algorithms assign forces among nodes for obtaining edges of balanced length (and as few crossings as possible). We refine the graphical layout by weighting the force between nodes through interaction-dependent factors. For instance, when laying out the program interaction graph, we amplify the force among two variables according to their number of joint occurrences in program constraints. Hence, roughly speaking, variables connected by a short edge have more such joint occurrences than variable connected by longer edges. These weights are calculated via a scoring similar to the MOMs heuristic (Pretolani 1996). Analogous weights are used for displaying the other aforementioned graph structures. This option does not provoke a complete re-structuring but helps exposing certain structures. For illustration, consider the program interaction graph in Fig. 2 where our graph layout leads to a state-wise clustering of variables.

Apart from furnishing different structural views, the purpose of interaction graphs is to offer *projection surfaces* for complementary information. A simple yet instructive such combination is *graph overlay*. The idea is to display the edges of one graph with the node layout of another. For instance, this allows us to study the interaction in learned nogoods in the context of the original problem structure. Another simple yet very effective combination is obtained by *graph coloring*.

[1] These graphs were used in (Sinz and Dieringer 2005) for visualizing SAT; cf. (Rish and Dechter 2000; Sinz 2007).

In fact, the data logger gathers several figures for each node during the solving process. Among these aggregated values on variables, we consider the number of decisions, number of conflict analyses, or the number of flipped values. These values can be aggregated over the whole solving process (by default) or any user-defined period of events (see *temporal aspects* below). The result is then projected through node colors on the interaction graph at hand. For this purpose, we use the color sequence from red to green. For instance in Fig. 2, we color variable nodes according to how often their values have been flipped. Accordingly, the variable having been flipped most often is colored in deep red, while 11 of 467 variables have been assigned only once and are thus in deep green. Notably, graph overlay and coloring can be freely combined. As mentioned, the coloring may reflect data collected during the entire or just a selected fragment of the solving process.

The inspection of node-specific information is supported by two complementary means. First, we integrate the aggregated values into the symbol table. Second, we offer a simple query language for filtering the displayed set of variable nodes. Finally, both capacities are dynamically linked to graph coloring and seamlessly adapt to changes triggered by the user in either of the three contexts.

A symbol table consists of four types of entries: a variable's solver identifier, its type, value, and symbolic representation. Each variable has a unique integer as solver identifier and may have one of three types: `atom`, `body`, or `atom/body`. The symbolic representation is type-specific: While a variable of type `atom` is associated with a unique ground atom, no representation is available for type `body`. The type `atom/body` represents multiple equivalent variables (eg. obtained through pre-processing) and gives all ground atoms associated with the solver identifier. The variable's value is mode-dependent and corresponds to the aggregated value used for coloring. Also, the (visible entries of the) symbol table can be sorted according to any attribute. For instance in Fig. 2, the node with the deepest red corresponds to the following entry.

id	type	value	symbol
26	atom/body	23	move(4,c,5)

Given that the coloring in Fig. 2 reflects the number of times that the value of the variable was flipped, the entry tells us that the atom `move(4,c,5)` was flipped 23 times.

The elements shown in the symbol table as well as the colored graph can be controlled via a simple query language. The language uses keywords `id:`, `type:`, and `val:` to refer to an entry's attributes; simple expressions (including wild card $*$) are used for matching symbolic representations. The keywords are followed by values of the respective type, except that `val:` additionally allows for simple range specifications of form $> i$ or $< i$ for some integer i. A conjunction is simply a blank; a disjunction is expressed by '`;`'. For instance in the context of Fig. 2, the query '`type:atom move(*) val:>4 val:<12`' selects all atoms formed from predicate `move` that have been flipped more than four and less than twelve times.

The interplay between the colored graph, the symbol table, and the query engine is designed to be highly dynamic. For instance, hovering over nodes in the graph centers the symbol table and highlights the corresponding entry. Selecting a cell in the symbol table produces the corresponding query and restricts coloring to the selected variables. And finally, posing a query selects the corresponding entries in the symbol table and restricts coloring accordingly. See Section 4 for more detailed information.

Temporal aspects. We capture temporal aspects by means of two-dimensional plots. While our interaction graphs provide views on the internal problem structure using aggregated node information, we use plots to provide insights into the dynamics of the solving process by exposing the development of its key figures. Instead of time, however, we use the sequence of conflicts, choices, or other events depending on the respective displayed aspect.[2]

The respective plots are enriched with the absolute and central moving average (over neighboring data points) as well as the median value. As an example, consider the left plot in Fig. 3, showing the length of each nogood in recording order. Accordingly, the x-axis is organized along conflicts (rather than choices). (The y-axis can be arranged in linear or logarithmic scale.) Alternatively, a frequency distribution can be provided, as done in the right plot in Fig. 3. The design of the interface was done to support the user in exploring the solving process. To this end, one may interactively select fragments of the plot and navigate through these fragments. Importantly, the selected solving spans may be used to build graphs reflecting the structural view and/or aggregated variable values collected in these segments.

4 The *clavis* and *insight* systems

In what follows, we describe the usage along with some implementation details of our data logger *clavis* and our visualization tool *insight*. Both systems are freely available as open source packages at (clavis). We illustrate the usage of *clavis* and *insight* through a small use-case. For this purpose, we consider the Towers of Hanoi problem from (Gebser et al. 2012).

clavis is a full-fledged ASP solver corresponding to *clasp* (2.1; (Gebser et al. 2012)) yet enhanced by data logging capacities. In fact, *clavis* is distributed as a patch to *clasp*, which facilitates its maintenance over progressing *clasp* versions. Apart from the obligatory name of the resulting database file, *clavis* allows for supplying a configuration file delineating the logged events along with the full set of (single-threaded) options of *clasp*. *clavis* also tolerates partial runs obtained either by *clasp*'s option '`--time-limit`' or user interrupts through SIGINT. For example, we may produce the event database `toh.h5` by the command (where `--heuristics=vsids` is an example *clasp* option):

```
gringo tohI.lp tohE.lp | clavis --heuristics=vsids toh
```

The resulting data is stored in an HDF5[3] database; it includes separate tables for each event type and a global index along with static data such as symbol tables. This approach allows for fast iteration of single event types as well as flexibility in logging. For instance, excluding some event types like propagations can significantly reduce the log size. Also, the logfile can be extended by additional events or meta data without breaking existing back-ends. The full documentation of the log format can be found at (clavis).

Different back-ends can be used for analyzing the logged information. Although it is possible to read the logfile directly via a library like *pytables*, we furnish an interface for sequential access and analysis through *clavis*' library *libclavis*. The benefits of this approach compared to direct access are much shorter read times due to caching and a simplified interface abstracting from the complexity of HDF5.

As an example, Listing 1 gives a simple *python* script extracting all learned nogoods from our example database `toh.h5`. This script relies on the *Listener* and *Player* interfaces of *libclavis*. The *Listener* supports data generation by implementing event handlers that process required

[2] For example, 2 on the x-axis refers to the second event.
[3] `http://www.hdfgroup.org/HDF5`

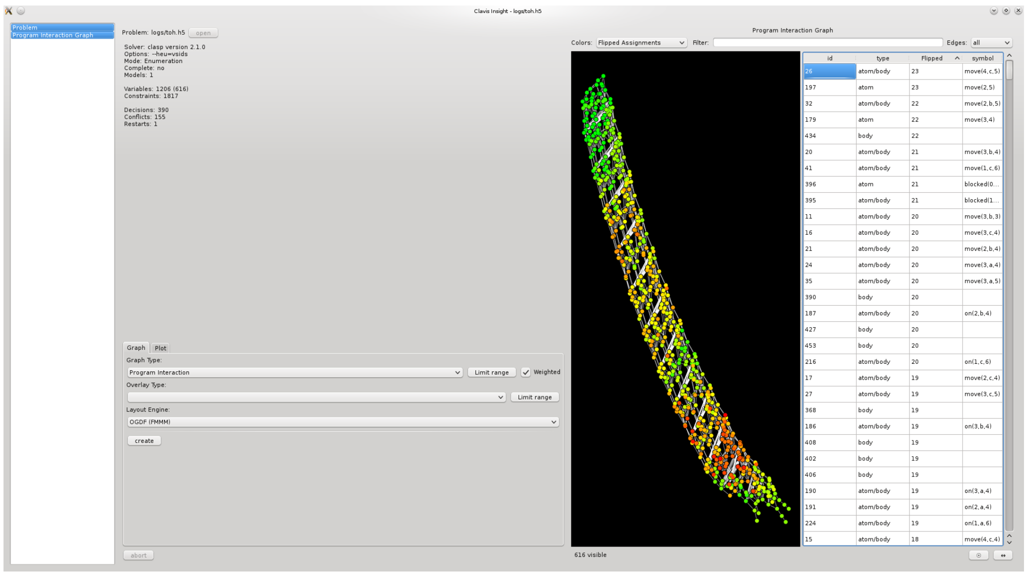
Listing 1. Extracting learned nogoods with libclavis (`eLib.py`)

```
from libclavis.logfile import Player, Listener

class LearnedPrinter(Listener):
    def event_learned(self, player):
        print ' '.join(map(str, player.event().lits()))

LearnedPrinter(Player('toh.h5'))
```

events. For instance, it allows for partial parsing of logfiles (by giving first and last index to read). The *Player* is able to test whether a logfile contains the events requested by the listener. For instance, these interfaces are used by *insight* to define several listeners that generate the data to be visualized. All generated data is stored using widely-used libraries such as *networkx* for graphs and *numpy* for sequential data. This allows for simple generation of derived data like centrality measures for graphs or frequency distribution for sequential data as well as the addition of more listeners.

insight is written in *python* and relies on open source graphics packages like *matplotlib* and *PyOpenGL* to visualize the data provided by *clavis*. It is launched either directly or by passing the event database, viz. `insight toh.h5`. After loading a logfile, *insight* displays the list of open views (and the *problem view*).

Let us explain some distinguished features of *insight* by looking at some screenshots in Fig. 2 to 4. On the left of all three, we see the *view list* giving all currently active (white on blue)


Clavis Insight - logs/toh.h5
Problem
Program Interaction Graph
open
Graph
Plot
Limit range
Limit range
create
abort
Program Interaction Graph

Problem: logs/toh.h5

Solver: clasp version 2.1.0
Options: --heu=vsids
Mode: Enumeration
Complete: no
Models: 1

Variables: 1206 (616)
Constraints: 1817

Decisions: 390
Conflicts: 155
Restarts: 1

Graph Type: Program Interaction [✓ Weighted]
Overlay Type:
Layout Engine: OGDF (FMMM)

Colors: Flipped Assignments
Filter:
Edges: all

id	type	Flipped	symbol
26	atom/body	23	move(4,c,5)
197	atom	23	move(2,5)
32	atom/body	22	move(2,b,5)
179	atom	22	move(3,4)
434	body	22	
20	atom/body	21	move(3,b,4)
41	atom/body	21	move(1,c,6)
396	atom	21	blocked(0...
395	atom/body	21	blocked(1...
11	atom/body	20	move(3,b,3)
16	atom/body	20	move(3,c,4)
21	atom/body	20	move(2,b,4)
24	atom/body	20	move(3,a,4)
35	atom/body	20	move(3,a,5)
390	body	20	
187	atom/body	20	on(2,b,4)
427	body	20	
453	body	20	
216	atom/body	20	on(1,c,6)
17	atom/body	19	move(2,c,4)
27	atom/body	19	move(3,c,5)
368	body	19	
186	atom/body	19	on(3,b,4)
408	body	19	
402	body	19	
406	body	19	
190	atom/body	19	on(3,a,4)
191	atom/body	19	on(2,a,4)
224	atom/body	19	on(1,a,6)
15	atom/body	18	move(4,c,4)

616 visible

Fig. 2. *insight* showing a program interaction graph with flipped assignments

and inactive (black on white) views; they can be (de)activated by mouse selection. The *problem view* in the middle of Fig. 2 is the default view after loading the logfile. At its top, it summarizes the key figures of the solving process at hand. Below, the actual visualization is configured and engaged, distinguishing the aforementioned structural ('Graph') and temporal views ('Plot').

The displayed setting allows for generating the program interaction graph. The resulting view is given on the right of Fig. 2. The graph nicely reflects the temporal structure of the Towers of Hanoi problem by grouping variables in a state-wise fashion. That is, variables with the same time stamp form clusters along the graph. The structure can be explored with zooming and panning functionalities (including reset button) and the visualization can be focused (while all other views are hidden). Note that the orientation of the graph is subject to random factors within the force-directed layout (eg. the graph might be mirrored for slightly varying data). However, the layout algorithm is seeded to assure that the same input leads to the same layout. Different layout engines can be used; currently, *insight* offers the choice between the *FMMMLayout* from *OGDF* and *sfdp* from *Graphviz*. The position of a variable can be inferred from the highlighting of the symbol table while hovering with the mouse over the node in the graph, or simply by searching for symbols via pattern matching (eg. for `move(*,1)`). The variable coloring in Fig. 2 reflects the number of flipped truth assignments. Inspecting the graph in conjunction with the symbol table reveals that the "hot-spot" of the problem concerns action and fluent variables with time stamps 3-6, changing their truth values more than twenty times (as shown in the symbol table). While these variables are in deeper red, the ones in the upper left part of the graph are colored in deeper green indicating that their truth value was rarely changed. This tells us that the goal conditions[4] are strong enough to fix the truth values of variables close to the final state. In fact, this can be made precise by posing a simple query like '`val:<5`' (similar to the left view in Fig. 4).

To complement this, let us look at the development of the conflict level[5] during the solving process in the left view in Fig. 3. To support this, we show on the right the frequency distribution

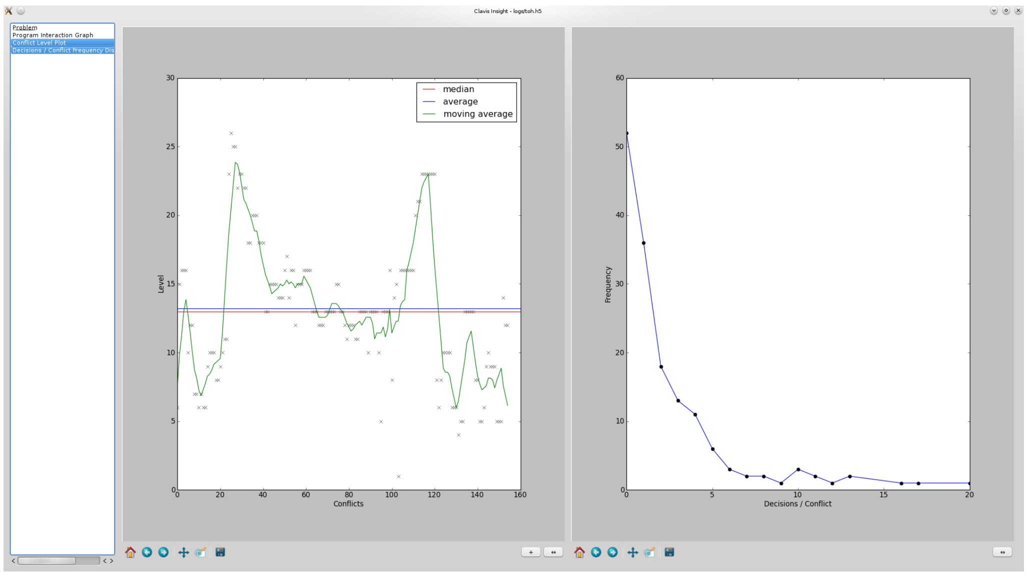
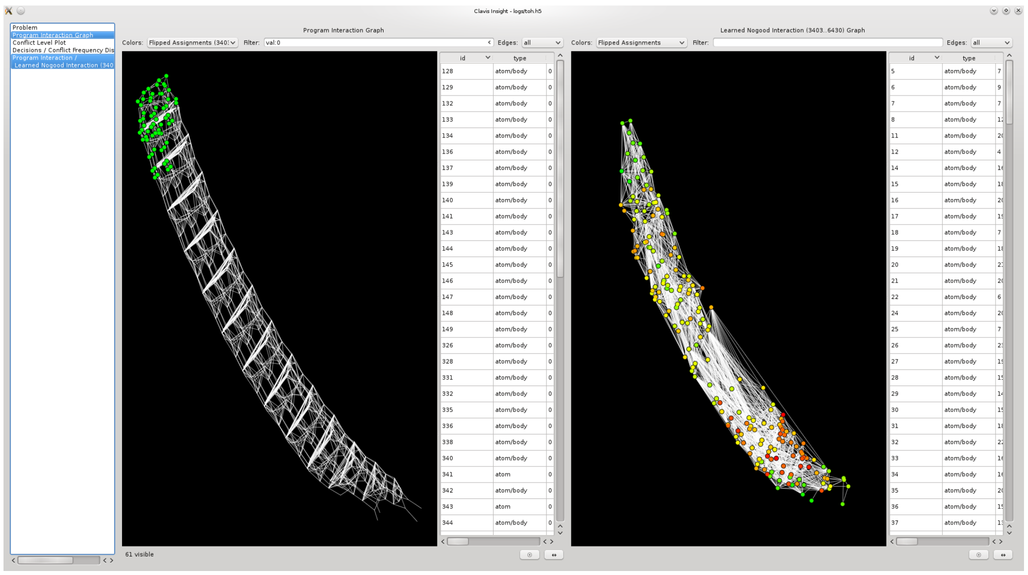


Fig. 3. *insight* showing conflict levels during the last two parts of the solving process and decisions per conflict

[4] That is, the goal state defined by `goal_on/2`.
[5] That is, the decision levels at which conflicts occur.

of the number of decisions per conflict;[6] this provides an idea on the progress made during the 155 conflicts on the x-axis on the left. For instance, 52 times no decision was made between two conflicts; here, the learned nogoods caused an(other) immediate conflict. 36 conflicts were obtained after a single decision. Now, following the (green) central moving average in the left view,[7] we observe two peaks dividing the solving process into three parts. Our hypothesis is that a part of the problem has been solved during the first solving phase. To explore this further, we take advantage of *insight*'s zooming and panning capacities. That is, via mouse control,[8] we select the last two segments of the solving process and generate (i) the program interaction graph colored according to the numbers of flipped assignments *during this span* along with (ii) the learned nogoods interaction yet projected on the program interaction graph colored in the same way (for comparability). The result is shown in Fig. 4. In fact, the coloring on the left is further


Clavis Insight - logs/toh.h5
Problem
Program Interaction Graph
Conflict Level Plot
Decisions / Conflict Frequency Dis
Program Interaction / Learned Nogood Interaction (340
Program Interaction Graph
Learned Nogood Interaction (3403..6430) Graph
61 visible

Colors: Flipped Assignments (340; Filter: val:0 Edges: all

id	type	
128	atom/body	0
129	atom/body	0
132	atom/body	0
133	atom/body	0
134	atom/body	0
136	atom/body	0
137	atom/body	0
139	atom/body	0
140	atom/body	0
141	atom/body	0
143	atom/body	0
144	atom/body	0
145	atom/body	0
146	atom/body	0
147	atom/body	0
148	atom/body	0
149	atom/body	0
326	atom/body	0
328	atom/body	0
331	atom/body	0
332	atom/body	0
335	atom/body	0
336	atom/body	0
338	atom/body	0
340	atom/body	0
341	atom	0
342	atom/body	0
343	atom	0
344	atom/body	0

Colors: Flipped Assignments Filter: Edges: all

id	type	
5	atom/body	7
6	atom/body	9
7	atom/body	7
8	atom/body	12
11	atom/body	20
12	atom/body	4
14	atom/body	16
15	atom/body	18
16	atom/body	20
17	atom/body	19
18	atom/body	7
19	atom/body	18
20	atom/body	21
21	atom/body	20
22	atom/body	6
24	atom/body	20
25	atom/body	7
26	atom/body	23
27	atom/body	19
28	atom/body	15
29	atom/body	14
30	atom/body	15
31	atom/body	18
32	atom/body	22
33	atom/body	16
34	atom/body	16
35	atom/body	20
36	atom/body	15
37	atom/body	13

Fig. 4. *insight* showing last two parts of the solving process

constrained by restricting the view to variables whose truth values remain unchanged during the last two parts of the solving process. This selection is accomplished via the simple query '`val:0`'. The resulting cloud of green nodes supports our conjecture that the truth value of the obtained variables has been fixed in the first part of the solving process. More evidence for this is provided by conflict learning because nothing has been learned about this program part during the considered span of events. This can be visualized by the projection of the learned nogood interactions onto the program interaction graph[9] on the right of Fig. 4: The green nodes in the left view do not appear in the right one. Hence, they do not appear in any nogoods learned in the last two phases of the solving process.

Important for the analysis of the ASP solving process is the consideration of larger problem

[6] That is, the number of decisions made since the previous conflict (or restart).
[7] This average considers the respective neighborhood of 1/20 of the whole dataset.
[8] Our manual selection picked the segment of events 3403 to 6430, as can be seen at the top of Fig. 4.
[9] That is, by keeping the layout of the variable nodes in the program interaction graph.

sizes. With only 1200 variables and 1800 constraints, the presented Towers of Hanoi example is a small problem compared to many real world examples. We employ a variety of techniques to handle larger problems by raising both the efficiency and flexibility of our tools. *clavis* stores the recorded data in a database designed to handle high volumes of data efficiently and in a format that is configurable to select the logged events. *libclavis* and *insight* are built upon libraries for efficient processing and presentation of information. This allows us to handle larger problems in the range of around 30,000 variables and 100,000 constraints for which the generation times of the different visualizations lie between a few seconds to two minutes on an Intel Core i7 processor. However, the specific runtimes are highly dependent on each specific problem.

5 Related work

There are several works using similar techniques in the area of SAT. Most influential to our work are *dpvis* and *3dvis* (Sinz and Dieringer 2005; Sinz 2007) as they also use interaction graphs to visualize the structure of SAT problems. Additionally, *dpvis* features online visualization through integration with a SAT solver for updating both the graph structure (e.g. for learned clauses) and colors (for assigned values) to reflect the current solving state. As mentioned above, our approach adapts the concept of representing program structure via interaction graphs and extends it to various types of interactions. Similarly, we also allow for the combination of program interaction graphs and learned nogoods but as part of a more general scheme involving multiple structural aspects of the solving process. Our offline visualization approach loses the interaction with the solver shown in *dpvis* but allows for much more freedom in combining and aggregating data from different parts of the solving process enabling the analysis of larger problems for which the approach of *dpvis* is impracticable. Furthermore, by including the symbol table of the ASP solver, we obtain a much deeper understanding of the problem's structure.

The SAT solver *CryptoMiniSat* (CryptoMiniSat) contains a logging feature and visualization that displays a variety of information similar to the temporal aspects shown in *insight*. The ASP solver *DLV* contains a tracer presented in (Calimeri et al. 2009).

6 Conclusion

We presented the design and implementation of a two-step approach for exploring ASP solving processes. Our data logger *clavis* provides a configurable tool for collecting data during ASP solving. Although *clavis* is necessarily system-specific, it relies on the state-of-the-art ASP solver *clasp*, which itself offers numerous configurations covering many different strategies to ASP solving. The data gathered by *clavis* is put in an easily accessible database format, viz. HDF5, that allows for an easy reuse by post-processing units. The most direct way to access this information is via scripting, as shown in Section 4. A more sophisticated way is furnished by our visualization tool *insight*. Apart from providing various structural and temporal views on the ASP solving process, *insight* is designed to foster the exploration of the solving process by providing the user with interactive means for changing the perspective on selected structures. Unlike previous approaches in SAT, ours greatly benefits from the availability of symbolic information in ASP. This allows us to provide a semantic link to the original problem representation. Clearly, there is yet much room for improvement and the modularity of our approach opens up many new possibilities for further analysis, profiling, and/or software engineering tools.

Acknowledgments. This work was partially funded by DFG grant GRK 1539/1.

References

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

CALIMERI, F., LEONE, N., RICCA, F., AND VELTRI, P. 2009. A visual tracer for DLV. In *Proceedings of the Second Workshop on Software Engineering for Answer Set Programming*, M. D. Vos and T. Schaub, Eds. Vol. 546. CEUR Workshop Proceedings (CEUR-WS.org), 79–93.

CLAVIS. `http://www.cs.uni-potsdam.de/clavis`.

CRYPTOMINISAT. `http://www.msoos.org/2013/04/cryptominisat-3-0-released`.

DECHTER, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Multi-threaded ASP solving with clasp. *Theory and Practice of Logic Programming 12,* 4-5, 525–545.

GOLDBERG, E. AND NOVIKOV, Y. 2002. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*. IEEE Computer Society Press, 142–149.

HACHUL, S. AND JÜNGER, M. 2004. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proceedings of the Twelfth International Symposium on Graph Drawing (GD'04)*, J. Pach, Ed. Lecture Notes in Computer Science, vol. 3383. Springer-Verlag, 285–295.

MARQUES-SILVA, J. AND SAKALLAH, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*. ACM Press, 530–535.

PRETOLANI, D. 1996. Efficiency and stability of hypergraph SAT algorithms. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. Johnson and M. Trick, Eds. Vol. 26. American Mathematical Society, 479–498.

RISH, I. AND DECHTER, R. 2000. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning 24,* 1-2, 225–275.

SINZ, C. 2007. Visualizing SAT instances and runs of the DPLL algorithm. *Journal of Automated Reasoning 39,* 2, 219–243.

SINZ, C. AND DIERINGER, E. 2005. DPvis - a tool to visualize the structure of SAT instances. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, F. Bacchus and T. Walsh, Eds. Lecture Notes in Computer Science, vol. 3569. Springer-Verlag, 257–268.

ZHANG, L., MADIGAN, C., MOSKEWICZ, M., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. ACM Press, 279–285.