A Visual Tracer for DLV

Francesco Calimeri, Nicola Leone, Francesco Ricca, and Pierfrancesco Veltri

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{calimeri,leone,ricca,pf.veltri}@mat.unical.it

Abstract. In software engineering, tracing is a specialized way for recording information about the execution of a program for debugging purposes. The more complex the system, the more difficult is developing a manageable, and thus practically useful, tracer. Answer Set Programming (ASP) systems represent no exception in this respect: the intrinsic complexity of reasoning required the design of elaborated evaluation algorithms.
In this paper, we present a suitable solution to the problem of tracing the execution of an ASP system and its implementation into the ASP system DLV. The tool herein presented features a graphical user interface and an on-line tracing method that puts it on the way between tracing and debugging. The range of applicability counts: bug fixing, system optimization, ASP-developing aids, and educational purposes.

1 Introduction

In software development, tracing is a specialized use of logging in order to record information about the execution of a program. This information is typically used by programmers for debugging purposes or (depending on the type and detail of information provided by the tracing system) by experienced system developers to diagnose problems or optimize implementations. Information provided by a tracing mechanism is usually employed by developers only, since usually there is not a standard output syntax, and the produced result might be either noisy (it might contain a lot of information which is useless for a specific purposes) or very long. Indeed, a common problem with tracing consists on the impossibility of isolating in a generic mechanism the information which is needed for detecting a specific problem, and thus a lot of useless information is inexorably printed.

Since the day of its first release the DLV system [1] is equipped with a simple tracing mechanism that is available to the developers; tracing instructions are instead removed by official release versions of DLV, for obvious optimization purposes. When enabled, this simple mechanism prints to the standard output a log of all internal system events and the value of some (relevant) internal variables. However, the unavoidable complexity of the algorithms employed for evaluating an ASP-program, and exploited by DLV, makes it quite difficult, or even impossible, to store and analyze such tracing output. It is worth noting that the evaluation of (non-ground) disjunctive ASP programs is a NEXPTIME^{NP} task [1, 2]; thus, the execution of an ASP system might produce a trace that is both inexorably large and difficult to handle, even in the case of small inputs.

In order to cope with this situation, we designed a general architecture for controlling and tracing the execution; we implemented such proposal into DLV, thus coming out with an advanced tracing system that has two important features: the execution of each task can be controlled (started, paused, or restarted), and the information produced can be set dynamically during the execution.

Our advanced tracer combines a graphical user interface and an on-line tracing method that puts it on the way between a tracing and a specialized system debugging tool.[1] The user can control the execution of the DLV system by means of suitable commands, and also ask the system to display some specific information, like e.g., the content of the internal data structures or the status of the system.

The resulting tool enjoys a wide range of applicability: bug fixing, system optimization, ASP-developing aids, and also educational purposes. Indeed, by following the trace of system execution, a program developer might analyze the behavior of the system and detect bugs or inefficient branches of the computation (as a by-product, a developer can be given an error-detection in the input specification). Moreover, by following the evaluation of a given encoding step-by-step, an ASP program developer might understand the reason for an inefficient evaluation, and tune its encoding for obtaining a more efficient of evaluation .

As already mentioned, the system features a graphical interface, that eases the interaction with the advanced tracing techniques; such interface makes the system suitable also for didactic purposes. Indeed, ASP systems act as a “black-box”, and it is not possible to see what is really happening inside. Conversely, professors can explain evaluation techniques by preparing and showing a live-demo of their actual implementation. Students might follow the execution, step-by-step, on their own machines; moreover, “playing” with the system, they can gain a more direct understanding of the working principles, which is facilitated by a clear image of what is going on “under the hood”.

Remark. It is worth noting that controlling and tracing the execution of an ASP system is quite a different task from debugging an ASP program. Indeed, the first task -the one addressed by the present work- aims at finding bugs and analyzing/controlling the behavior of an ASP system (which is a piece of software usually written in some imperative language); whereas, on the other hand, debugging an ASP program has the purpose of finding bugs in an logic program (which consists of ASP-language code to be then evaluated by means of an ASP system). While scanning the trace produced by the ASP system, the developer could also find errors within the logic program in input; but this kind of error-detection in the input specification is not the main purpose of our tracer. Techniques and tools specifically devised for debugging ASP programs (see e.g., [4–8]), are more appropriate than the tool herein presented for the second task. Note also that analogous considerations do not hold for the tracing-based debuggers for Prolog systems (see, e.g., [9]), where, due to the operational nature of the semantics of the supported language, tracing the execution results to be very useful also for debugging the logic program in input.

[1] This tool provides a method for tracing and debugging ASP-systems; the interested reader can find details on debugging techniques for ASP programs in [3].

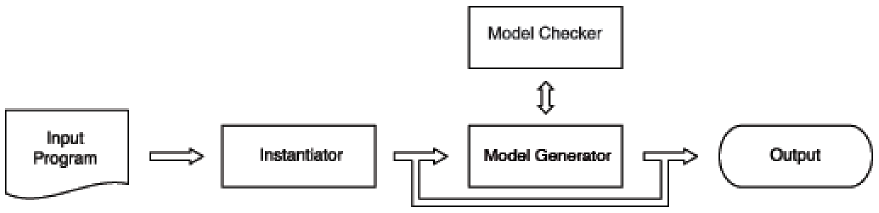


Fig. 1. DLV architecture.

The remainder of the paper is organized as follows: in Section 2 we describe the architecture of the DLV system; in Section 3 we describe our advanced tracing system; finally, we describe the system usage and the graphical user interface in Section 4.

2 The DLV System Architecture

We now outline the general architecture of DLV, which is schematically reported in Figure 1. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system. In general, an input program $\mathcal{P}$ contains variables, and the first step of a computation of an ASP system is to eliminate these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$. This variable-elimination process is called *instantiation* of the program (or *grounding*), and is performed by the *Instantiator* module (see Figure 1). A naïve Instantiator would produce the full ground instantiation $Ground(\mathcal{P})$ of the input, which is, however, undesirable from a computational point of view, as in general many useless ground rules would be generated. DLV therefore employs sophisticated techniques which are geared towards keeping the instantiated program as small as possible. A necessary condition is, of course, that the instantiated program must have the same answer sets as the original program. Moreover, if the input program is normal and stratified, the DLV Instantiator is able to directly compute its stable model (if it exists). The subsequent computations, which constitute the non-deterministic part of an ASP system, are then performed on $ground(\mathcal{P})$ by both the *Model Generator* and the *Model Checker*. Roughly, the former produces some "candidate" answer set, whose stability is subsequently verified by the latter. The Model generator of DLV implements a backtracking search algorithm, similar to a DPLL procedure of SAT solvers, which works directly on the ground instantiation of the input program. As previously pointed out, the *Model Checker* verifies whether an answer set candidate at hand is an answer set for the input program. This task is solved in DLV by calling a specialized procedure, since it is as hard as the problem solved by the Model Generator for disjunctive programs, while it is trivial for non-disjunctive programs.[2]

Finally, once an answer set has been found, DLV prints it in text format, and possibly the *Ground Reasoner* resumes in order to look for further answer sets. Note that, other

[2] However, there is also a class of disjunctive programs, called Head-Cycle-Free programs [10], for which the task solved by the Model Checker is provably simpler, which is exploited in the system algorithms.

traditional ASP-systems basically agree on the same general architecture even if they employ different techniques for implementing the same system components.

In sum, the evaluation of an ASP program in DLV can be divided in three *main tasks:* Instantiation, Model Generation, and Model Checking. Each of them requires to be *traced* for debugging purposes, and in the following Section we describe how our advanced tracing mechanisms deals with this requirement.

3 Advanced Tracing for DLV

The old DLV tracing mechanism is simple: it prints to standard output a log of all the internal system events, followed by the value of some relevant internal variables (e.g. current partial interpretation, current rule to be processed, etc.). More in detail, each module of DLV has a specialized set of traced variables and events, and the detail of the information produced can be set statically, for each module, to a given level ranging from 0 to 3 (minimum and maximum level roughly correspond to tracing-disabled and full information, respectively). This information allows for reconstructing an entire DLV execution, and/or to focus on the details of a single (or some) task, like e.g. Instantiation.

The main problem of this tracing system is however very easy to be seen: even small inputs can produce a huge tracing output, which might be either very difficult to be handled or even impossible to be stored in the file system. Indeed, tracing information is noisy, in the sense that the developer cannot isolate in a generic tracing mechanism the information which is needed for detecting a specific problem, and a lot of useless information is inexorably printed. Moreover, the unavoidable complexity of the algorithms employed for solving each single task of ASP-program evaluation can make even impossible to store and analyze the tracing output. Note that, Instantiation is in general EXPTIME-hard (the produced ground program being potentially of exponential size with respect to the input program), and both the Model Generator and the Model Checker implements a backtracking procedure that might require to execute (and, thus trace) an exponential number of operations (w.r.t. the size of the ground instantiation of the program!).

In order to cope with this situation, we designed and implemented in the DLV system a general architecture for controlling and tracing the execution, that has two important features: the execution of each task can be controlled (started, paused, or restarted) and the information to be printed can be set dynamically during the execution.

In Figure 2 is depicted the general architecture of our tracing method. In particular, the system is able to receive and recognize a sequence of commands in XML format from the standard input (or from a given file); those commands are recognized by the *command parser* module and inserted in a *command queue*. Each evaluation task of the system has been modified in order to stop periodically its normal execution in some predefined *breakpoints*, pick-up a new command from the queue and execute it. Commands might require to: set the system events to be printed, print the value of some status variable or the content of some internal data structure (e.g. one might ask whether some atom is true or false in the current partial interpretation); to continue the execution up to the next breakpoint; to undo the execution of some task; to terminate the process;

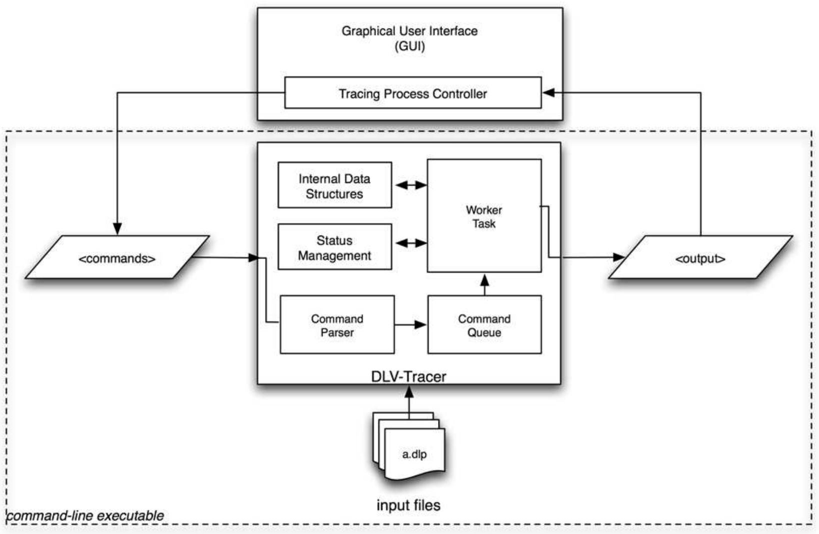


Fig. 2. Tracing Architecture.

etc.[3] The result of each command, together with the output generated by the system are printed to the standard output according to a specifically designed XML format. In this way, one can control dynamically the execution, and select the information it needs. Thus, the information to be printed is dramatically reduced, and the user can focus on the information regarding a specific moment of the execution. It is worth noting that, we carefully placed several breakpoints in the evaluation algorithms; breakpoints, that can be enabled of disabled by setting the *grain* level of execution, e.g. in the Model Generator one can decide to stop the execution at each choice point (lowest level of grain) or at the end of each propagation rule (finest level of control). The level of grain itself can be set dynamically by exploiting a specific command.

The DLV system enriched with this advanced tracing can be controlled either manually (by writing the commands from the console) or by exploiting a graphical user interface that is described in the next Section.

4 System Usage and Graphical User Interface

This section describes the usage of the herein presented system, then illustrates the Graphical User Interface (GUI) that has been conceived in order to ease interaction.

Commnad line interface. The tracing system is embedded into the DLV system, thus it features a command-line interface; data are exchanged through standard input/output

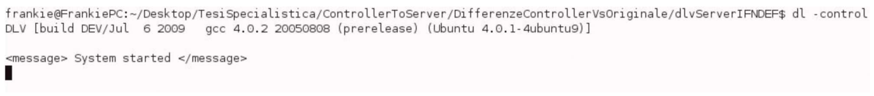
[3] For a complete listing of the available commands see Appendix 4.

streams. The tracer can be started by invoking DLV with ”$-control$” option, and an optional XML input file containing a list of commands:

```
\$./dl -control [commandFile]
```

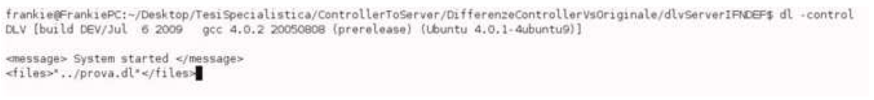
If input file is not provided, then the system awaits for commands on the standard input. We now report the snapshots of the command-line debugger while running on an example program.

Suppose now that we want to trace the Model Generator, initially, we start the DLV system with the option - control:



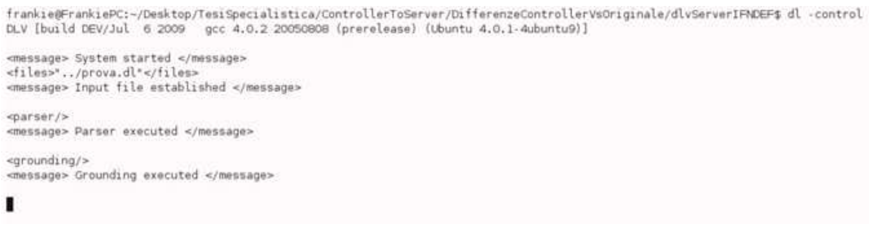
```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
```

The logic program we will give in input to DLV is stored in the file *prova.dl*, and contains the following rules: $a \lor b.\ c \lor d.\ e :- a.$. The image below shows how to set this program as input for DLV: just type the <*files*> tag.



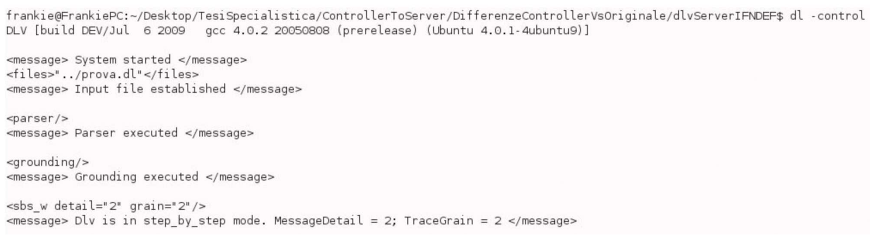
```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
```

Obviously, one might set more than one file by repeating the same command. Once the input is set, we can start the parser and then the Instantiator as follows:



```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>
```

To enable tracing we set the tracing mode by inserting the <*sbs_w/*> command. The tag has two attributes: detail and grain. The detail level determines (as in the old tracing method) the quantity of information to be printed; whereas, the grain level determines the number of active breakpoints, respectively. In the following we set both Trace and Grain levels to two:



```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>

<sbs_w detail="2" grain="2"/>
<message> Dlv is in step_by_step mode. MessageDetail = 2; TraceGrain = 2 </message>
```

Then, we run the Model Generator, the system executes a part of the computation and then DLV stops at the first breakpoint (enabled for this level of grain) and prints the tracing log. Then, we go to the next breakpoint with the command <*sbs_n/*>. In this case we will see the answer sets found written between braces (see Figure 3)

At each breakpoint we can modify the tracing configuration by adding or removing the information to be printed. In this example we require to add some additional information to the log (see Figure 4).

Going forward, Model generator finishes, and DLV waits for commands. In this example we reset the grounding of the program and then we restart both Grounding and Model Generator requiring to visualize the answer sets of the program without any tracing (see Figure 5).

Alternatively, the user might start the debugging session by exploiting the graphic interface.

Graphical User Interface. The GUI allows the user to exploit the full power of DLV Controller and Advanced Tracer in a simpler and more intuitive fashion. In the following, we describe how he interface is structured.

On the left of the main window is the management area for DLV input. On top of this area, a tree structure represents the part of the file system of the machine running DLV (Figure 7). The user can choose the root of such tree while starting the application, but if it can also be modified later.

Below the tree structure there is the list of the files given as input to DLV for current session (Figure 8).

A *session* is initialized when the system starts and it closes when the DLV process ends; a session can also be forced to close by the user through an appropriate button.

The central area of the main window is divided into two parts: the main available commands and a tracing management area are placed in the upper side, while the lower consists of a "Console".

The tracing management area features some fields that remember the overall status of the application.

Fields are in charge of showing: the current status of DLV (Figure 10), the last command given by the user (Figure 11), DLV options that are currently enabled (Figure 12), and the grain and detail levels set during the last tracing analysis (Figure 13).

Under these fields a table, initially empty, is placed which show all the information printed by the Advanced Tracer during the session (Figure 14).

As described above, the information printed at each breakpoint changes according to the detail level; however, the user can also customize the current configuration by means of appropriate buttons.

In order to facilitate a better understanding of what is happening during the session, the part of the information displayed which is updated by the Tracer at each breakpoint is highlighted in red; the rest is gray (Figure 15).

For each piece of information name and current value are available. If a value is too long, it can be entirely displayed by clicking the "Enlarge" button.

The "Manage Info" button allows to customize the current display configuration for the information printed by the Tracer.

The GUI will show only the pieces of information explicitly included in the first list of Figure 16; nevertheless, such list can be customized by adding other pieces of information from the second list, or by removing currently selected pieces of information.

Finally, a "Console" is showed in the lowest area of the window (Figure 17). This text-area shows the output of the Controller as it is released; thus, when a command is invoked, the user can view the results.

5 Conclusion

In this paper we have presented an advanced tracing methodology which has been especially conceived for controlling and monitoring the execution of an ASP system.[4] We have implemented it on the DLV system, and developed a Graphical User interface that allows to manage tracing operations in a friendly environment.

The advanced tracing technique herein presented can be fruitfully exploited by system developers, with the purpose of finding bugs or optimizing the execution of internal algorithms. Moreover, tracing can be exploited by ASP program developers in order to optimize (and, in some cases, fix) input ASP programs: indeed, by following the trace of system execution, the ASP program developer might better understand the behavior of the exploited ASP system when a specific encoding is evaluated, and thus provide an alternative (hopefully more-efficiently-evaluable) encoding, or fix an incorrect one.

Thanks to its friendly interface, the advanced tracer might also be employed for didactic purposes; indeed, students can discover what is going on "under the hood", thus better understand underlying techniques and evaluation algorithms.

References

1. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
2. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425

[4] Even though each ASP system features its own algorithms and techniques (and thus, also peculiar variables an data structures), the idea of controlling the execution by means of proper breakpoints and an external graphical interface which deals with the system by means of an XML syntax can be easily adapted and implemented into ASP systems different from DLV.

3. De Vos, M., Schaub, T., eds.: SEA'07: Software Engineering for Answer Set Programming. Volume 281., CEUR (2007) Online at `http://CEUR-WS.org/Vol-281/`.
4. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. TPLP **9**(1) (2009) 1–56
5. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A Meta-Programming Technique for Debugging Answer-Set Programs. In: AAAI'08, AAAI Press (2008) 448–453
6. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 86–100
7. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning, Lake District, UK (2006) 77–84
8. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
9. Roychoudhury, A., Ramakrishnan, C.R., Ramakrishnan, I.V.: Justifying proofs using memo tables. In: PPDP. (2000) 178–189
10. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87

A Appendix: Tracing Commands

In the following we report some of the most important tracing commands that allow to customize the tracing.

Customizing the Configuration Once the tracing mode has been set, or during the analysis phase, the information to be printed can be customized according the user's need. This is done by means of the following statements.

– `<show_info_tracer/>`
  shows all information concerning current configuration; this will be printed every time an active breakpoint is reached. The short version of this statement is `<show_t/>`.
– `<add_info_to_tracer>"infoName1;`...; `infoNameN"</add_info_to_tracer>`
  add a piece of information to the current tracing configuration; information will be printed at each step from now on. The information to be added must be written between start tag and end tag. A hort version of this statement is available as `<a_to_t>"infoName1`... `infoNameN"</a_to_t>`.
– `<delete_info_of_tracer>"infoName1;`...; `"infoNameN</delete_info_of_tracer>`
  removes a piece of information from the current tracing configuration. The short version is `<d_of_t>"infoName1` ... `infoNameN"</d_of_t>`.
– `<empty_info_of_tracer/>`
  empties the current tracing configuration; this will force the tracer to print no information at all. The short version is `<e_of_t/>`.

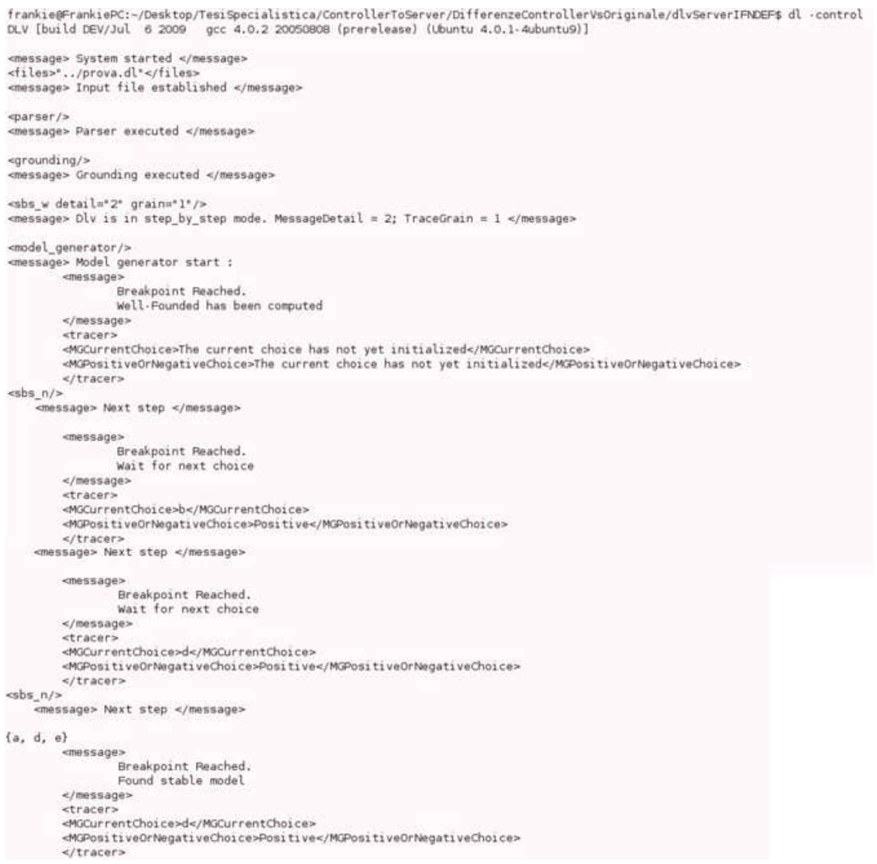
Breakpoints Commands When DLV stops, at any breakpoint, the user can inspect and trace the execution by means of the following commands:

- `<step_by_step_next/>`
  allows to exit the current breakpoint and go forward to the next one. Short version is `<sbs_n/>`.
- `<step_by_step_continue/>`
  allows to leave the Tracing mode; hence, DLV goes ahead until the end of computation with no stops (thus ignoring any other breakpoint). The short version is `<sbs_c/>`.
- `<step_by_step_view>"InfoName"</step_by_step_view>`
  this command prints information "on-demand". Indeed, the user can ask the Tracer to print some pieces of information which are not contained in the current configuration. Short version is `<sbs_v>"InfoName"</sbs_v>`.

There are also some commands which are defined as "special"; these can be invoked by the user only at some specific breakpoints.

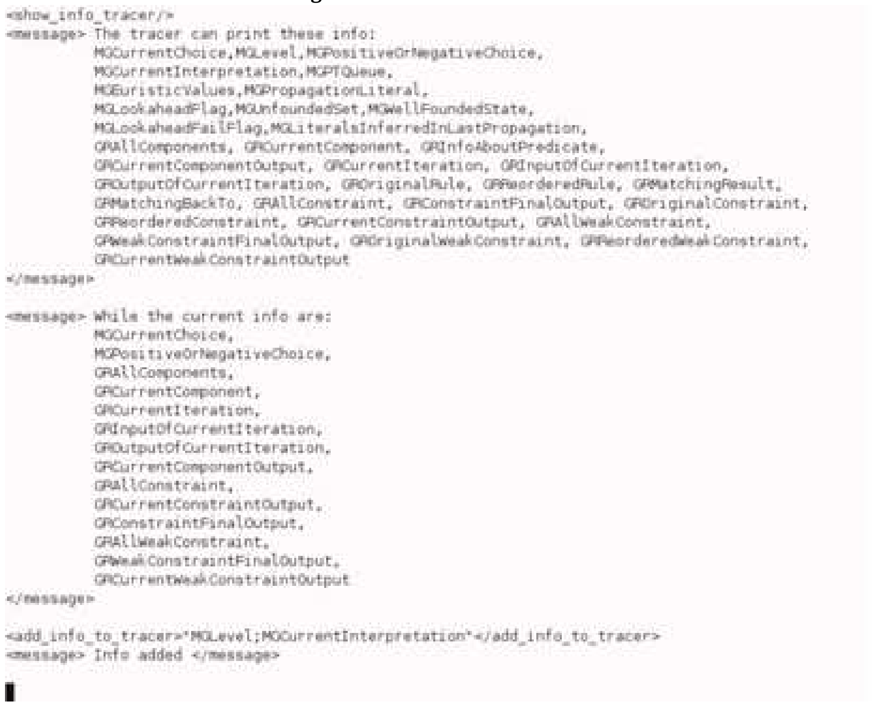
- `<step_by_step_go_back>"X"</step_by_step_go_back>`
  or
  `<sbs_gb>"X"</sbs_gb>`
  forces DLV to go back of X level; if L is the current level, the computation starts over from level: $L - X$. This command can be invoked only while the Model Generator is being traced; in particular, only at a breakpoint where DLV checks the stability of the current model, or when it is waiting for the next choice.
- `<step_by_step_stop_when>"X"</step_by_step_stop_when>`
  or
  `<sbs_sw>"X"</sbs_sw>`
  forces DLV to go ahead without any stop while the atom X is not true; once the atom X becomes true, the system will stop at next breakpoint. This command can be executed at any breakpoint, both during the Grounding or the Model Generation phases.
- `<step_by_step_go_component>"X"</step_by_step_go_component>`
  or
  `<sbs_gc>"X"</sbs_gc>`
  forces the system to go ahead and stop just before the evaluation of the component X has to start. It can be executed only during while the Grounding is traced, in particular only during the evaluation of the components of the input program.
- `<step_by_step_go_rule>"X"</step_by_step_go_rule>`
  or
  `<sbs_gr>"X"</sbs_gr>`
  forces the system to go ahead and stop just before the evaluation of the rule X has to start. It can be executed only while the Grounding is being traced, in particular during the evaluation of the components of the input program. This command has effect only if the grain level is set to 2: indeed, there are no stops at rule level with a lower grain level.

– `<step_by_step_go_constraint>"X"</step_by_step_go_constraint>`
  or
  `<sbs_gcn>"X"</sbs_gcn>`
  force the system to go ahead and stop just before the evaluation of the constraint X has to start. It can be executed only during while the Grounding is being traced, in particular during the evaluation of the components or the constraints of the input program. This command has effect only if the grain level is set to 2: indeed, there are no stops at constraint level with a lower grain level.
– `<step_by_step_go_wconstraint>"X"</step_by_step_go_wconstraint>`
  or
  `<sbs_gwcn>"X"</sbs_gwcn>`
  forces the system to go ahead and stop just before the evaluation of the weak constraint X has to start. It can be executed only while the Grounding is being traced, in particular during the evaluation of the weak constraints. This command has effect only if the grain level is set to 2: indeed, there are no stops at weak constraint level with a lower grain level.



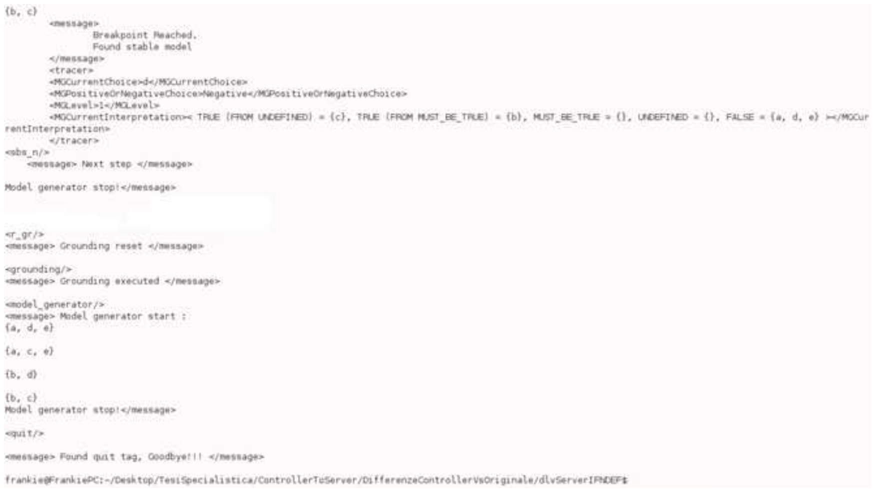
```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>

<sbs_w detail="2" grain="1"/>
<message> Dlv is in step_by_step mode. MessageDetail = 2; TraceGrain = 1 </message>

<model_generator/>
<message> Model generator start :
        <message>
                Breakpoint Reached.
                Well-Founded has been computed
        </message>
        <tracer>
        <MGCurrentChoice>The current choice has not yet initialized</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>The current choice has not yet initialized</MGPositiveOrNegativeChoice>
        </tracer>
<sbs_n/>
    <message> Next step </message>

        <message>
                Breakpoint Reached.
                Wait for next choice
        </message>
        <tracer>
        <MGCurrentChoice>b</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
    <message> Next step </message>

        <message>
                Breakpoint Reached.
                Wait for next choice
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
<sbs_n/>
    <message> Next step </message>

{a, d, e}
        <message>
                Breakpoint Reached.
                Found stable model
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
```

Fig. 3. Run the Model Generator.



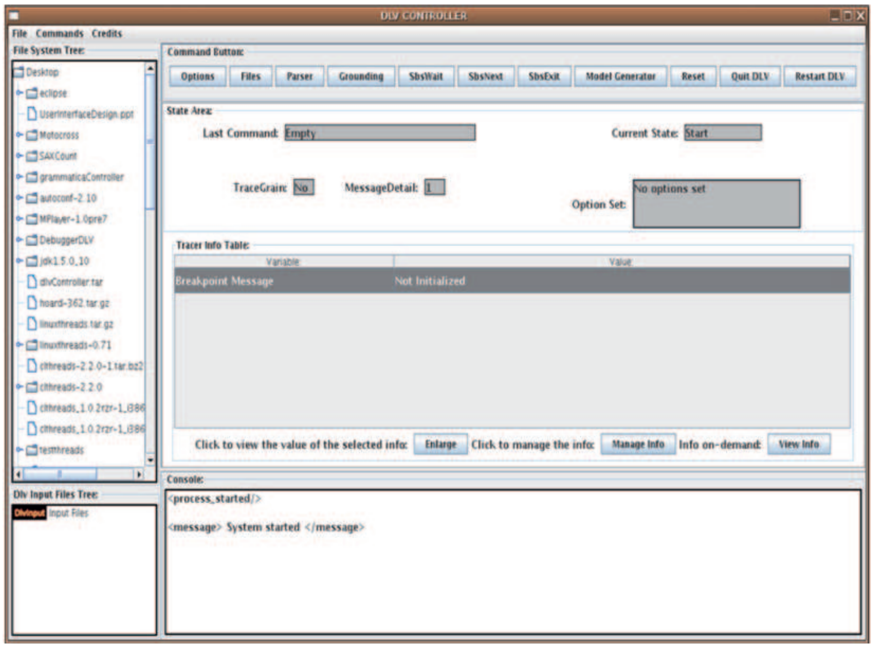
```
<show_info_tracer/>
<message> The tracer can print these info:
             MGCurrentChoice,MGLevel,MGPositiveOrNegativeChoice,
             MGCurrentInterpretation,MGPTQueue,
             MGEuristicValues,MGPropagationLiteral,
             MGLookaheadFlag,MGUnfoundedSet,MGWellFoundedState,
             MGLookaheadFailFlag,MGLiteralsInferredInLastPropagation,
             GRAllComponents, GRCurrentComponent, GRInfoAboutPredicate,
             GRCurrentComponentOutput, GRCurrentIteration, GRInputOfCurrentIteration,
             GROutputOfCurrentIteration, GROriginalRule, GRReorderedRule, GRMatchingResult,
             GRMatchingBackTo, GRAllConstraint, GRConstraintFinalOutput, GROriginalConstraint,
             GRReorderedConstraint, GRCurrentConstraintOutput, GRAllWeakConstraint,
             GRWeakConstraintFinalOutput, GROriginalWeakConstraint, GRReorderedWeakConstraint,
             GRCurrentWeakConstraintOutput
</message>

<message> While the current info are:
             MGCurrentChoice,
             MGPositiveOrNegativeChoice,
             GRAllComponents,
             GRCurrentComponent,
             GRCurrentIteration,
             GRInputOfCurrentIteration,
             GROutputOfCurrentIteration,
             GRCurrentComponentOutput,
             GRAllConstraint,
             GRCurrentConstraintOutput,
             GRConstraintFinalOutput,
             GRAllWeakConstraint,
             GRWeakConstraintFinalOutput,
             GRCurrentWeakConstraintOutput
</message>

<add_info_to_tracer>"MGLevel;MGCurrentInterpretation"</add_info_to_tracer>
<message> Info added </message>

▮
```

Fig. 4. Add tracing information.



```
{b, c}
        <message>
                Breakpoint Reached.
                Found stable model
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Negative</MGPositiveOrNegativeChoice>
        <MGLevel>1</MGLevel>
        <MGCurrentInterpretation>< TRUE (FROM UNDEFINED) = {c}, TRUE (FROM MUST_BE_TRUE) = {b}, MUST_BE_TRUE = {}, UNDEFINED = {}, FALSE = {a, d, e} ></MGCur
rentInterpretation>
        </tracer>
<sbs_n/>
    <message> Next step </message>

Model generator stop!</message>



<r_gr/>
<message> Grounding reset </message>

<grounding/>
<message> Grounding executed </message>

<model_generator/>
<message> Model generator start :
{a, d, e}

{a, c, e}

{b, d}

{b, c}
Model generator stop!</message>

<quit/>

<message> Found quit tag, Goodbye!!! </message>

frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$
```

Fig. 5. End tracing.



Last Command: Empty

Current State: Start

TraceGrain: No

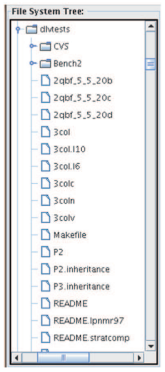
MessageDetail: 1

Option Set: No options set

Variable	Value
Breakpoint Message	Not Initialized

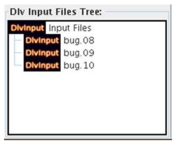
```
<process_started/>

<message> System started </message>
```

Fig. 6. GUI: Starting interface.



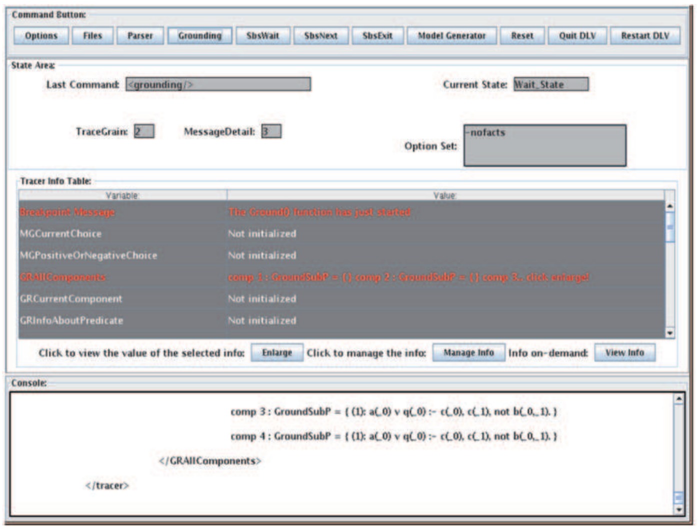
```
File System Tree:
dlvtests
    CVS
    Bench2
    2qbf_5_5_20b
    2qbf_5_5_20c
    2qbf_5_5_20d
    3col
    3col.l10
    3col.l6
    3colc
    3coln
    3colv
    Makefile
    P2
    P2.inheritance
    P3.inheritance
    README
    README.lpnmr97
    README.stratcomp
```

Fig. 7. GUI: Current State of DLV.



```
Dlv Input Files Tree:
DlvInput Input Files
    DlvInput bug.08
    DlvInput bug.09
    DlvInput bug.10
```

Fig. 8. GUI: Last Command Executed.



Command Button:

State Area:

Last Command: <grounding/>

Current State: Wait_State

TraceGrain: 2

MessageDetail: 3

Option Set: -nofacts

Tracer Info Table:

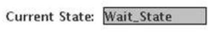
Variable	Value
Breakpoint Message	The Ground() function has just started
MGCurrentChoice	Not initialized
MGPositiveOrNegativeChoice	Not initialized
GRAllComponents	comp 1 : GroundSubP = { } comp 2 : GroundSubP = { } comp 3... click enlarge!
GRCurrentComponent	Not initialized
GRInfoAboutPredicate	Not initialized

Click to view the value of the selected info: Click to manage the info: Info on-demand:

Console:

```
                    comp 3 : GroundSubP = { (1): a(_0) v q(_0) :- c(_0), c(_1), not b(_0,_1). }
                    comp 4 : GroundSubP = { (1): a(_0) v q(_0) :- c(_0), c(_1), not b(_0,_1). }
            </GRAllComponents>
    </tracer>
```

Fig. 9. GUI: Main Area of the Window.



Current State: Wait_State

Fig. 10. GUI: Current State of DLV.

Last Command: <grounding/>

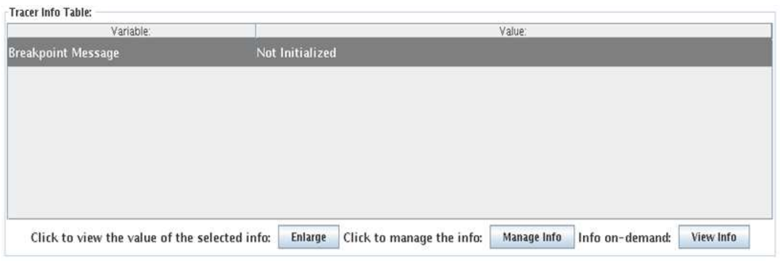
Fig. 11. GUI: Last Command Executed.

Option Set: -nofacts

Fig. 12. GUI: DLV Options Enabled.

TraceGrain: 2 MessageDetail: 3

Fig. 13. GUI: Grain and Detail Levels.

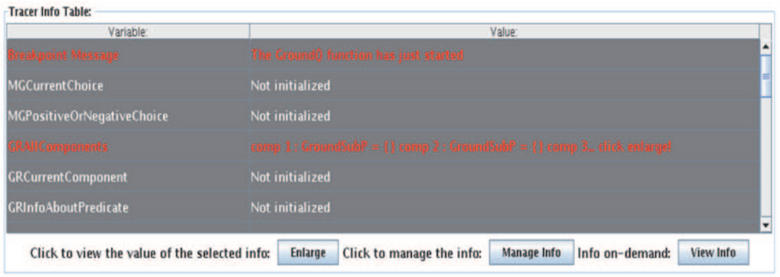


Tracer Info Table:

Variable:	Value:
Breakpoint Message	Not Initialized

Click to view the value of the selected info:
Enlarge
Click to manage the info:
Manage Info
Info on-demand:
View Info

Fig. 14. GUI: Tracing Table at the Beginning.

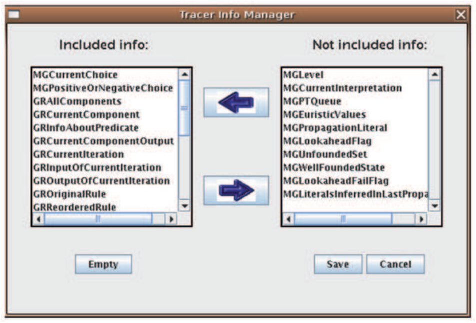


Tracer Info Table:

Variable:	Value:
Breakpoint Message	The Ground() function has just started
MGCurrentChoice	Not initialized
MGPositiveOrNegativeChoice	Not initialized
GRAllComponents	comp 1 : GroundSubP = { } comp 2 : GroundSubP = { } comp 3... click enlarge!
GRCurrentComponent	Not initialized
GRInfoAboutPredicate	Not initialized

Click to view the value of the selected info:
Enlarge
Click to manage the info:
Manage Info
Info on-demand:
View Info

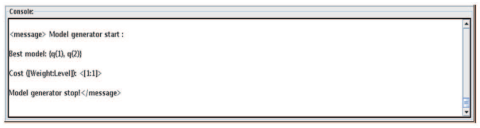
Fig. 15. GUI: Tracing Table during Analysis.



Tracer Info Manager

Included info:	Not included info:
MGCurrentChoice	MGLevel
MGPositiveOrNegativeChoice	MGCurrentInterpretation
GRAllComponents	MGPTQueue
GRCurrentComponent	MGEuristicValues
GRInfoAboutPredicate	MGPropagationLiteral
GRCurrentComponentOutput	MGLookaheadFlag
GRCurrentIteration	MGUnfoundedSet
GRInputOfCurrentIteration	MGWellFoundedState
GROutputOfCurrentIteration	MGLookaheadFailFlag
GROriginalRule	MGLiteralsInferredInLastPropa
GRReorderedRule	

Fig. 16. GUI: Dialog for Information Management.



Console:

```
<message> Model generator start :

Best model: {q(1), q(2)}

Cost ([Weight:Level]): <[1:1]>

Model generator stop!</message>
```

Fig. 17. GUI: Console.