The DiSCiPl Project: Debugging Systems for Constraint Programming

(Invited presentation)

September 24, 1997

DiSCiPl is an ongoing european project on Constraint Logic Programming Environments (Project Nb 22523). It started October 1rst 1996 and is a 2.5 years project. Here is a brief overview of the project.

More information can be found at URL: http://discipl.inria.fr/

The consortium is composed of 4 industrial members COSYTEC(F), PrologIA(F), ICON(I) and BIS (B), as well as 4 academic members: INRIA(F), University of Bristol(UK), Linköping University(S) and University of Madrid (E). There is also an advisory committee, composed of the PMG (UK) and several European companies and establishments using CP technology (from E,S,P,D,I,F).

Project Manager:
Pierre Deransart,
INRIA-Rocquencourt,
BP 105, F-78153 Le Chesnay Cedex, France
Tel: +33 1 39635536, Fax: +33 1 39635330
Email: Pierre.Deransart@inria.fr
URL: http://www.inria.fr

Manpower: 21 man/year.

1 General Presentation

The goal of the proposed project is to define, to implement and to assess novel and effective debugging systems for Constraint Programming.

Debugging is considered in a broad sense: it concerns both validation aspects (to build a correct application) as well as methodological aspects (to find the best solution to a problem by better understanding of constraint solver behaviour).

To satisfy these objectives, tools must locate and explain bugs, and graphic tools must help interpreting program behaviour and results. Existing tools reveal to be ineffective in most industrial situations and tools developed for imperative or functional programming are not adapted to the context of CP. The main reasons are that the huge numbers of variables and constraints makes the computation state difficult to understand, and that the non deterministic execution increases drastically the number of computation states which must be analysed. Existing tools do not include all aspects of debugging is a single environment.

The expected results of the project include:

- novel techniques of high level debugging,
- integration of these novel techniques into industrial CP platforms,
- assessment of these techniques in industrial applications,

The taken approach is based on three paradigms currently used in advanced debugging:

- Declarative debugging: The clear semantics of CP languages allows to adapt debugging techniques developed in the context of Logic Programming.
- Assertion based methods: The user may be unable to understand the results or the behaviour of a program, due to the complexity of the constraint system in some intermediate computation state. We propose methods such as the instantiation of the answer constraints and the use of assertions describing the expected properties.
- Graphics based methods. The use of graphical tools is of great interest both to display properties of the solutions, and to follow the history of changes of variables or constraints. They may be used during debugging, at several levels, for constraint display and update and combined with assertion based methods.

None of these paradigms individually provides a complete solution to the CLP debugging problems. The orientation taken by the project is firstly to consider them separately in order to improve and adapt them in the context of CLP, when existing works are not sufficient, then to find ways and to experiment in order to combine them in powerful and useful debugging tools. Plugging together basic tools derived from each of these paradigms is the most interesting challenge of the project.

The definition of the debugging tools has been elaborated in two steps.

First, debugging needs have been analysed with a questionnaire distributed among partners and their clients, international related mailing lists, and made public on the project WEB site. This first step confirmed that two crucial aspects of debugging of CP applications are performance debugging and correctness debugging. Tools should in particular provide efficient help to observe and understand complexity of resolution.

In a second step possible tools have been analysed and classified. It has been observed that tools alone could not be sufficient and needed to be completed with a programming method aimed at facilitating debugging and thus called "DiSCiPl debugging methodology". We give some details on the two steps.

2 Debugging Needs

We think we can collapse the multiple emerging needs of debugging in two main categories:

- correctness debugging;
- performance debugging.

The rest of this section is devoted to illustrate the characteristics of these two categories.

2.1 Correctness debugging

By correctness debugging we mean the actions the programmer does in order to make the program behave as expected, i.e. according to his/her intention. The ultimate goal of this debugging activity is to locate all the errors and faults which are revealed by missing/incorrect answers (semantics errors). Note that the missing answers problem resolves itself into two formidable (and undecidable) problems: detecting the reason of failure and detecting non termination.

As questionnaires showed, the potential sources of missing and incorrect answers are: typos in the program text, wrong typing, wrong design and coding of modules and procedures, wrong constraint modelisation. Finding these errors is the traditional debugging activity, but in the CP framework this is somehow complicated by the fact that the execution of the program is data-driven. Hence traditional step by step, source oriented and trace oriented approaches are insufficient.

Sometimes, however, the problem is not in the program itself but in the consistency of the data set on which it must operate. If we include in the semantics of the program all the possible data sets on which it is intended to operate, we can see how data-inconsistency can be seen as a special case of correctness debugging: the semantics of the data are inconsistent or incompatible with the semantics of the program. It is worth noting that this time it is not the program which must be debugged but the data-set. However the symptom is likely to be the same: a missing or incorrect answer, and in general an unexpected behaviour of the program. Whatever the tools we are using for debugging the semantics of the program, they should end up to be helpful to locate that the problem is on the data (although fixing it or guarantee data consistency is another issue).

2.2 Performance debugging

Performance debugging has been posed as a very relevant problem. Constraint programs must be not only correct but also efficient. This is very often a primary requisite for a constraint program. As stated in M. Meyer, "Debugging Constraint Programs, TR ECRC-95-15":

Developing CLP applications is a difficult task. This is to a large extent due to the fact that CLP is usually being applied to combinatorial search problems for which no efficient algorithms are known. Instead of following some well-known rules, the users have to experiment with various models, approaches and strategies to solve the problem. Apart from handling the usual correctness debugging of their programs, users are also facing the problem of performance debugging, i.e. finding a way to execute CLP programs efficiently. To date, there exists no satisfactory methodology for debugging CLP programs. There are basically two ways to approach the problem: either try to apply all available methods exhaustively, last resort being the simplification and downscaling of the problem , or to analyse the behaviour of the program and try to understand what is the reason for its poor performance. The current CLP systems unfortunately offer little support for this task and there are not widely available tools which would support tracing and (performance) debugging CLP programs.

Most of the times, bad performances are due to a poor comprehension and modelisation of the problem. Unfortunately this aspect is sometimes obscured by the constraint solver. As a first consequence of the problem of performance debugging, there is a need to understand constraint propagation. Points to investigate are if and why

- constraints are not waked when expected;
- constraints are uselessly waked (small or no reduction of domains).

However the problem it is not only to understand how propagation works, but also how propagation fails. Finding the reason of failure is useful in trying to obtain efficient pruning of the failing search alternatives. Performance debugging rises also the need to understand the search space traversal. Again, finding very quickly a first set of solution and then running a long time for new ones is a very common behaviour for a CP optimisation program execution. The programmer is left with the doubt if the program is lost in a local search or if the search space does not contain any better solution. Even when he/she realises that the first case applies, very few information is provided on how he/she could change the labelling strategy.

3 Debugging Tools

The purpose of this section is to respond to the needs presented in Section 2, and briefly describe tools which will be realised by the DiSCiPl project.

For performance debugging, tools will be mainly based on visualisation of the program behaviour. Several tools will offer the possibility of studying the program behaviour from different points of view in order to help the user to understand the solvers behaviour and to improve the search strategies. For correctness debugging, tools will take advantage of the declarative nature of CLP, based on a declarative kernel language. Such kernel has a "declarative semantics" independent from execution and performance. Our purpose is to exploit such semantics as much as possible to debug CLP programs; in particular to apply the methods of static analysis of programs. The tools proposed will lead to a high level of integration of debugging phases, combining performance and correctness debugging.

Debugging environments will be developed to facilitate integration of the tools and their use for various purposes. For example unexpected answers may trigger unexpected search, in which case it may be convenient to interleave the use of tools based on the declarative semantics with the use of search visualisation tools.

According to the requirement expressed in the previous sections, graphical tools are perceived by users as potentially very useful, offering an interesting complement to assertion-based and declarative debugging tools. One of the main reasons for this is an intuition that for some aspects graphical tools can potentially depict complex information in a more compact and understandable form. This may allow coping with information which is more detailed and/or more procedural than the one the user can deal with using assertion-based tools, as well as having a global or more abstract picture of certain complex data or computation.

The tools to be developed should be useful to a wide range of users. This includes naive users, which are mainly concerned with a basic understanding of the operation of the constraints program and of correctness issues. However, such users are often also interested in a basic understanding of some performance issues. The tools should also be interesting to advanced users, which will be developing large applications, and which will be more concerned with performance issues and the operational details of the search. Finally, system developers represent yet another group of users, who are basically interested in studying the performance of internal algorithms. However, although the tools developed will probably also be of use to system developers, they are not perceived as the primary target of the project's tools.

Due to the variety of users and aspects apparent from the previous discussion, the range of possible tools for debugging of constraint programs is quite large. Many of the aspects to be dealt with are quite experimental, and the literature on the topic is relatively scarce. This implies some difficulty in defining a priori the characteristics of the debugging and visualisation tools to be developed in a very precise way. Some of these aspects are user requirements for debugging, controlling, understanding and improving constraint programs developed for complex and combinatorial use applications.

We propose two kinds of tools: those referring to a single computation, and those referring to all the computations of a program. In both kinds graphical interfaces may be of great value.

3.1 Tools related to a single computation

They support investigation of a single test run, i.e. of the search space for some initial goal. They include:

1. graphical tools for presentation of the search space, variables and constraints to the user, mainly for purposes of performance debugging. The tools should provide various means of focusing and abstraction, in order to cope with the, usually great, size of the search space, number of variables and constraints. This will include as a special case the functionalities present in box model Prolog debuggers;

2. declarative debugger(s) for localisation of incorrectness and insufficiency errors (incorrect and missing answers). Declarative diagnosis is a particular kind of searching an SLD-tree; a proof tree used in incorrectness diagnosis can be seen as an abstraction of a branch of an SLD-tree. In traditional approaches the LD tree traversed is hidden from the user, who has to concentrate on declarative aspects. A graphical interface may visualise the nodes of the tree connected with the queries of the diagnoser.

3.2 Tools referring to all possible computations

A program showing an unexpected behaviour violates some requirement. Finding the error may be quite difficult since this may require exhaustive testing. The tools proposed in this section address a more general question: whether or not all computations of a given program satisfy a given requirement. These tools facilitate analysis of a single computation while searching for the reasons of unexpected behaviour. In this approach, a program analysis is done statically and its results are valid for all runs of the program. Here we plan tools for both inferring and checking program properties, mainly for the purpose of correctness debugging. The properties will be expressed as assertions, the proposal focuses on the use of types. Note that program analysis may concern declarative and dynamic properties as well. Notice also that the language of assertions may be useful as a focusing tool for graphical presentation of the search space, for example for localisation of certain nodes.

It is possible that static analysis techniques could also be applied to performance debugging. This would require some kind of global cost analysis for all computations of the program. This may be a subject of future research, but in the present state of the art it is rather difficult to propose a tool for performance debugging based on global analysis techniques. Verification can in general be done by proving a given property (requirement) via a dedicated prover. Another approach to verification is to infer automatically properties of a program and to compare them with the requirements. Independently of the technique used, verification can be seen as a process which receives a program and a set of requirements as input and returns, for each requirement, one of the following three values:

- the requirement holds, i.e. it is verified. This is the most favourable situation;
- the verifier is unable to prove nor disprove the requirement. This may be because the requirement is not fulfilled or because the verifier is not complete;
- the requirement is proved not to hold, i.e. there exists a computation for which the property does not hold. This is a "symptom" that something is wrong in the program and diagnosis should start in order to correct it.

We therefore have three types of tools:

- provers: tools which attempt to prove that a certain property holds. Take as input a program and a property. Give as output the result "proved", "disproved", or "unknown";
- analysers: tools which infer properties from a program. Take as input a program. Give as output a set of properties;
- comparators: tools which compare two (sets of) properties in an attempt to prove one of them, the "requirement", with the other one, the "warrant". Take as input the two properties. As in the case of provers, they give as output the result "proved" (if the requirement is implied by the warrant), "disproved" (if the requirement is guaranteed not to hold in view of the warrant), or "unknown".

3.3 Target platforms

During the project, a series of debugging tools which combine several basic tools as sketched above will be developed. The better understood will be developed by the industrial partners, and the more experimental will be prototyped by the university partners. Four CLP platforms will be upgraded with such debugging environments:

- CLP(FD + Interval) at INRIA;
- CIAO at UPM;
- CHIP V5 (beta version) at COSYTEC;
- Prolog IV (beta version) at PrologIA.