1. Debugging of Constraint Programs: The DiSCiPl Methodology and Tools

P. Deransart[1], M. Hermenegildo[2], and J. Maluszynski[3]

[1] INRIA-Rocquencourt, Projet LOCO, BP 105, F-78153 Le Chesnay Cedex, France. `Pierre.Deransart@inria.fr`
[2] Facultad de Informática, Universidad Politécnica de Madrid, 28660-Boadilla del Monte, Madrid, Spain. `herme@fi.upm.es`
[3] Linköping University, Department of Computer and Information Science, S 581 83 Linköping, Sweden. `janma@ida.liu.se`

This introduction gives a general perspective of the debugging methodology and the tools developed in the ESPRIT IV project DiSCiPl *Debugging Systems for Constraint Programming.* It has been prepared by the editors of this volume by substantial rewriting of the DiSCiPl deliverable *CP Debugging Tools* [1.1].

This introduction is organised as follows. Section 1 outlines the DiSCiPl view of debugging, its associated debugging methodology, and motivates the kinds of tools proposed: the assertion based tools, the declarative diagnoser and the visualisation tools. Sections 2 through 4 provide a short presentation of the tools of each kind. Finally, Section 5 presents a summary of the tools developed in the project. This introduction gives only a general view of the DiSCiPl debugging methodology and tools. For details and for specific bibliographic references the reader is referred to the subsequent chapters.

1 The DiSCiPl View of Debugging

The work performed in the project has addressed two main categories of CP debugging:

– *correctness* debugging, and
– *performance* debugging.

Correctness debugging aims at locating program constructs that cause that the computed answers are different from those expected. In particular this concerns syntax errors, wrong answers, missing answers, mode violations, non termination, and several kinds of unexpected behaviour. Detection of inconsistencies in input data is another topic of correctness debugging.

Performance debugging aims at identification of the reasons for poor efficiency of computations, such as inadequate labelling strategy or poor constraint propagation. The overall goal of performance debugging is to shorten the time for finding feasible and/or optimal solutions.

The subsequent sections summarise the approaches pursued in DiSCiPl for addressing correctness debugging and performance debugging.

1.1 Correctness debugging in DiSCiPl

DiSCiPl pursued two different approaches to correctness debugging:

– static debugging,
– debugging based on run-time symptoms

Static Debugging. *Static debugging* aims at finding errors without executing the program. In DiSCiPl static debugging is based on static analysis of the program and on automatic checking of assertions.

Automatic checking of assertions may be able to prove correctness of the program with respect to formally specified requirements concerning certain properties of the program that should hold in *all* computations. For example, a formal requirement may specify call and success patterns of a predicate in all computations of the program. If a program is not correct w.r.t. a given specification, any attempt to prove it will fail.

Static analysis techniques make it possible to infer automatically certain properties of the program. The inferred properties may or may not conform to user expectations, or to a priori given specification describing the expectations. In the latter case the discrepancy between the inferred and the expected/specified properties is called an *abstract symptom*. Thus, abstract symptoms may be found manually, by inspection of the results of static analysis, or automatically, by comparison of the assertions describing expected properties with those generated by static analysis. Existence of an abstract symptom shows that the program is not correct w.r.t. the specification of the expected properties.

The process of locating a construct causing the abstract symptom will be called *static diagnosis*. This construct will also cause a failure in any attempt to prove correctness of the program with respect to the specification of expected properties, and can be located in such an attempt.

A crucial issue in static debugging is how to design a language for expressing specifications and results of static analysis. The language should on one hand allow expressing complex properties such as those inferred by static analysis and on the other hand should be easy to understand by the user. The statements of the language will be called *assertions*. Thus, assertions should provide a basis for two-way communication between the user and the static debugging tools.

It should be noticed that the use of assertions is not restricted to static analysis and correctness proofs. In some DiSCiPl tools they are also used as tests checked during the execution of the program or for selection of particular data for inspection by the user during execution visualisation.

Debugging based on run-time symptoms. This approach concerns the case when in some execution the program behaves in some way which is different from that expected by the user. Such a discrepancy between the expected and the actual behaviour of a program in a single computation is called a *run-time symptom*. Debugging based on run-time symptoms aims at finding the cause of a run-time symptom that occurred in a single execution. In the case of correctness debugging the symptom can usually be linked to an atom, a predicate or a clause. This may be achieved by analysis of the computation where the run-time symptom appears. A run-time symptom, such as a wrong answer or a missing answer, can also be characterised in terms of declarative semantics. Thus, at least for some run-time symptoms, debugging can be based either on the declarative semantics or on the operational semantics. Both possibilities have been pursued in DiSCiPl.

An approach to debugging based on declarative semantics known as *declarative diagnosis* [1.12] makes it possible to locate errors responsible for wrong and missing answers. The method has been further developed in DiSCiPl.

On the other hand, some of the DiSCiPl visualisation tools (designed primarily for performance debugging) facilitate analysis of single computations, hence debugging based on operational semantics.

1.2 Performance debugging

Poor performance is often caused by modelling of the problem in an inadequate way or by adopting an inadequate search strategy. Often, such an error cannot be identified just at the level of a single predicate or constraint. Unfortunately, there are no general methods for finding reasons for poor performance. Therefore, it is necessary to provide tools which help the user to understand (very) complex computations. Such tools should be able to present a complex computation at various levels of abstraction in a form easy to understand by the user. Three aspects of particular importance are:

– presentation of the control of the execution,
– presentation of the constraint store at different points in the execution
– presentation of global constraints.

Tools must also allow to re-do or modify computations at some point with the purpose of exploring different strategies.

The DiSCiPl approach to performance debugging was based on the guidelines listed above. Several visualisation techniques proposed and implemented in the project will be surveyed below and will be discussed in more detail in separate chapters. It should be noted that these are general tools which help the user to understand the execution of the program, and thus, in addition to performance debugging, they may also be of interest for correctness debugging.

1.3 The DiSCiPl Debugging Methodology and Tools

We now present a general view of the proposed DiSCiPl debugging methodology and show how it is supported by the tools developed in the project.

The proposed debugging methodology consists of (possibly interleaved) successive steps of correctness debugging and performance debugging. Location of an error results in modification of the program. The new version may still be subject to debugging.

The proposed methodology is illustrated in Figure 1. At each stage of the program development process we have an actual version of the program, (possibly) some input data and a (possibly empty) specification expressed in the assertion language.

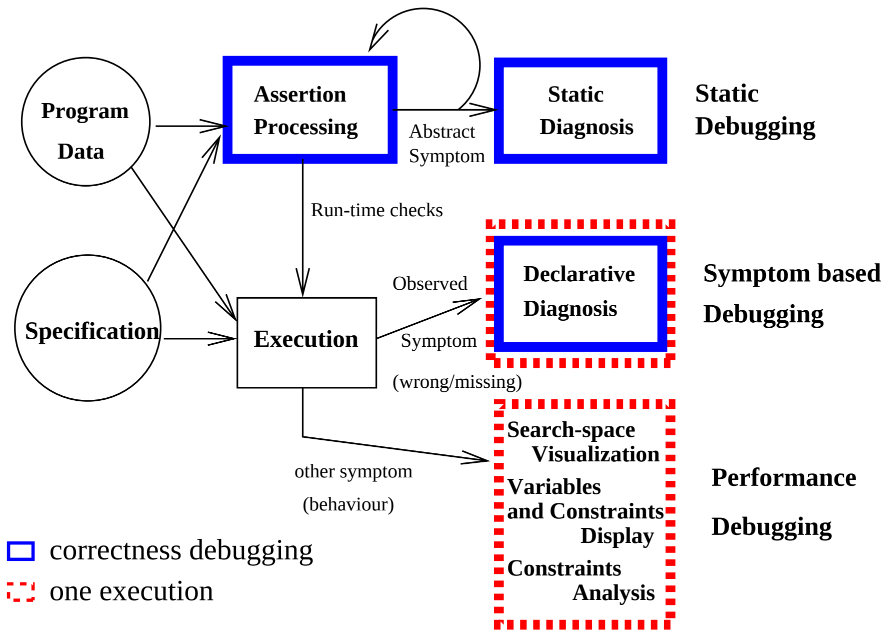

Fig. 1. The DiSCiPl Debugging Methodology

The underlying debugging methodology is as follows. The first objective is, understandably, program correctness. Correctness is dealt with at a first stage at compile-time. We distinguish here the *assertion processing* phase and the *static diagnosis* phase.

Two different approaches to the assertion processing phase have been used in DiSCiPl. The first one is an extension of the classical verification techniques to the case of constraint logic programs. In this approach, a complete specification of the program is required and verification aims at actually proving that the program satisfies such specification. Failure in proving the specification may locate program constructs which are responsible for bugs. The second approach is based on first inferring properties of the program by means of static analysis and then comparing such properties with the existing specification. If the inferred properties are incompatible with some part of the specification then an abstract symptom is found. The advantage of this second alternative is that a complete specification of the program is not required, though the more complete such specification is, the more bugs can be automatically detected.

In case abstract symptoms are discovered in the assertion processing phase, they should be corrected before executing the program. The cyclic arrow in the assertion processing box indicates that this is an iterative process. After correcting the program, assertion processing must be performed again in order to check whether abstract symptoms still remain in the corrected version of the program. Though the assertion processing tools often identify the cause of an abstract symptom, it is also convenient to have automatic tools for static diagnosis, as indicated in the figure. The intention is to detect bugs as early as possible, i.e., during compilation or even editing. This can only be achieved by (semi-) automatic analysis of the (not necessarily completely developed) program in the presence of some (approximate) specifications. An example of such techniques is type checking, which has long been proved to be useful for this purpose. Our approach introduces a framework for working with properties that will be more general than classical type systems.

The fact that no more abstract symptoms are detected does not imply that no bugs are left in the program. At this point, program testing in the classical sense has to be performed. However, the methodology also aids the programmer in this respect. During the assertion processing phase, programs may optionally be automatically annotated with run-time checks for those requirements which turned out to be impossible to prove in the assertion processing phase. This will allow detecting further violations of the specifications during program execution on sample input data. The result is detection of run-time symptoms. These have to be diagnosed and the detected errors have to be corrected. Since run-time tests introduce substantial overhead in program execution, the annotation with run-time checks is performed only upon user's request and possibly temporarily, i.e., only until the behaviour of the program can be assumed to be correct.

Once correctness is sufficiently established, it is time for performance debugging. The visualisation tools developed in the project make it possible to present abstract views of a single computation. In particular, they will be used to examine those computations that are inefficient in finding expected solutions.

Figure 1 gives a general idea about the tools needed to support the proposed methodology. They include tools like assertion processors, static diagnosers, declarative diagnosers, and various kinds of visualisers. Several tools of this kind have been implemented in the project on various platforms. They are surveyed in the forthcoming sections and discussed in more detail in the remaining chapters of the book.

2 The assertion-based tools

The static debugging tools developed in DiSCiPl are based on assertions. We first briefly discuss the language of assertions and then the static debugging tools based on it.

2.1 The language of assertions

Assertions are used for describing selected aspects of the intended or/and the actual behaviour of the program. Thus, they specify some requirements about all answers or computations, or they describe some properties that are satisfied by all answers (or by all computations) of the program.

There is a trade-off in the design of an assertion language: the more expressive the language of assertions is, the more subtle properties of programs can be described. However, this also causes increased difficulties in verification and/or inference of such subtle properties. Also, a complex assertion language could be difficult to understand by the user.

The choice of a suitable assertion language was an important design decision that provided a common basis for development of various tools on different platforms. An extensible assertion language has been designed and is in use in several tools developed in the project. The language is described in more detail in Chapter 1. Here, we outline only some principles of its design and we give a few examples of assertions.

There are two main kinds of assertions, those which relate to predicates, called *predicate assertions* and those which relate to program points (*program-point assertions*). Predicate assertions can be classified into three categories: (1) assertions describing success states of a predicate, sometimes called *post-conditions*, (2) assertions describing call states of a predicate, sometimes called *preconditions*, (3) assertions describing properties of the whole computation of a predicate (a sequence of states).

Predicate assertions link with a predicate some properties which should hold, respectively, in all call states, all success states or in all computations. For example, the assertions:

```
:- calls p(X,Y) : ground(X).
:- success p(X,Y) => list(Y).
:- comp p(X,Y): (ground(X), var(Y)) + not_fail.
```

state, respectively, that in all calls to `p`, the first argument must be ground (i.e., a term without variables), in all success states for `p` the second argument must be a list, and that every call to `p` with the first argument ground and the second a variable should not fail.

The language of assertions is extensible in the sense that its definition does not restrict a priori the properties to be referred to by the assertions. Properties can be predefined predicates of the language (like `ground` and `var` above) but can also be defined by the user. Chapter 1 discusses general aspects of using the DiSCiPl assertion language for program validation and debugging.

2.2 The Tools

We now discuss briefly the DiSCiPl tools supporting static debugging. They include: the Generic Preprocessor for CLP Debugging and its two instances CiaoPP (for the Ciao Prolog system) and CHIPRE (for CHIP), described in Chapter 2, the Prolog IV assertion tool (Chapter 3), and the static type-based diagnoser of CHIP discussed in Chapter 4 which has been also ported to Calypso.

We assume that a program is given together with a (possibly empty) set of assertions describing some required properties of the program (to be called *check assertions*).

Ideally, we would like the assertions to be automatically *checked*. That is, we would like to have a tool that receives a program and a set of check assertions as input and returns, for each of the assertions, one of the following three values:

- **proved:** the requirement holds in every computation. This is the most favourable situation. It indicates that the requirement is verified.
- **disproved:** the requirement is proved not to hold in some computations. This means that something is wrong in the program and we should locate the program construct responsible for that.
- **unknown:** the tool is unable to prove or disprove the requirement. If this may happen the tool is called *incomplete*.

Such automatic checking of assertions is one of the basis for static debugging in DiSCiPl. Though the tools used are incomplete, in many cases they are able to prove or to disprove the check assertions provided by the user. For a disproved set of check assertions, the constructs responsible for incorrectness of the program are identified in the static diagnosis process. Even in "don't know" cases it is often possible to locate fragments of the programs where violation of the check assertions is not excluded. This is often a useful warning.

It is often the case that no check assertions are provided, or the verification of the provided ones gives a "don't know" answer. The DiSCiPl static debugging tools address this problem in different ways.

The principle of the preprocessors (CiaoPP, CHIPRE) is fully automatic operation that for a given program and (possibly empty) set of check assertions finds as many abstract symptoms and locates as many errors as possible. The preprocessor uses *abstract interpretation* [1.5] to infer actual properties of a given program which are then written in terms of assertions. Automatic inspection of such inferred assertions allows detecting irregularities, like empty types of predicates, which are often good indicators of bugs in the program. The tool also compares the inferred assertions with the existing check assertions and with the assertions describing relevant properties of the library predicates and reports the discovered abstract symptoms. This sometimes may be sufficient to locate errors and generate warnings. For an assertion which cannot be proved or disproved run-time tests may be included in the program. It is important to mention that the preprocessor may detect a good number of bugs without any user-provided check assertions.

In the Prolog IV assertion tool the focus is on verification of check assertions. In order to use it, one has to augment the program with check assertions describing expected call states and success states of all program predicates. The assertions are to be constructed from a small set of basic properties including atomic constraints of Prolog IV and a few other properties. The tool attempts to verify the assertions and reports on the errors located during the verification and also on missing and inconsistent assertions. Run-time tests are inserted for assertions which were neither proved nor disproved during the verification. In contrast to the preprocessors this tool is not able to handle programs without check assertions.

The type-based diagnoser is an interactive tool for locating causes of abstract symptoms. The diagnosis is done by searching for verification failures, thus using a principle similar to the above discussed tool. An important difference is that the check assertions needed are not assumed to be given a priori but are interactively requested "by need". The process is started and controlled by a discovered abstract symptom. The session is preceded by static analysis of the program. The analyser infers types of all predicates. They may be inspected by the user and the session starts when one of them is identified as an abstract symptom. (Alternatively a symptom can be obtained by comparison of the inferred types with check assertions, if the latter are given). The requests of the diagnoser concern expected types of program predicates. They may be answered by referring to the types inferred or by providing different ones.

3 The declarative diagnoser

Declarative diagnosis is a technique for locating errors in a program on the basis of run-time symptoms that appear in a single terminating computation. The diagnosis session may be started if the outcome of a terminated computation is a symptom. The key idea is that the analysis of the computation is partly automated. The system queries the *oracle*, which is normally the user, about intermediate results of the analysed computation. The oracle has to decide whether the results shown at each step of the diagnosis are symptoms or not. In addition, it has to tell the system what would be the expected result if the one shown is a symptom. Oracle answers are used by the diagnosis algorithm to control the search for the error.

It should now be clear why the diagnosis is called declarative: the queries concern only the results, that is, they refer to the declarative semantics of the program, while the operational aspects of the analysed computation are hidden from the oracle. This allows to locate errors in the program without having to deal with control aspects, which are hidden in the diagnosis algorithm.

The symptoms dealt with by a declarative diagnosis divide naturally in two categories:

– an *incorrectness symptom* is a wrong answer. More precisely, the computed answer for a given goal g is a constraint c. It is an incorrectness symptom if c has some solutions not expected by the user.
– an *incompleteness symptom* is observed if some specific solution expected by the user is not given by any of the computed answers.

Declarative diagnosis of each kind of the symptom is done by different algorithms.

The declarative diagnoser developed in DiSCiPl is described in Chapter 5. The main difficulty with the declarative diagnosis technique is that the intermediate results may be difficult to understand by the human oracle. The visualisation and abstraction techniques developed in the project may be helpful in this respect.

4 Visualisation Tools

Sometimes the tools discussed so far cannot detect the source of an incorrect or missing answer in a program, or the program shows an unsatisfactory performance whose source cannot be detected by static analysis-based tools. The latter behaviour is usually intimately related with the operational semantics of the language, and cannot be uncovered by tools based on declarative semantics. In the former case, sometimes an extensive exploration of the runtime behaviour of the program (possibly with the aid of some intuitive, high-level representation of such behaviour) is of great help, allowing the programmer to grasp the problems present in the execution. In this section we will present tools intended to be used when programs produce wrong results for a given test data or, in general, when their performance falls below acceptable limits. They are based on the operational semantics of the language, and highlight different aspects of the computation in order to help the user to understand the reasons for the undesired behaviour.

Most visualisation and runtime debugging tools are based on representations of the execution profile of constraint programs using a Prolog-like selection rule, and thus depict an LD tree. However, classical debuggers (i.e., those customarily used for imperative languages, and even for Prolog) fall altogether short when it comes to CLP, and new approaches are needed: the implicit selection rule, the use of backtracking, the constraint propagation, and the (encapsulated) labelling strategies of CLP have to be taken into account and somehow reflected in the representation of the execution, for in many cases the culprit of an unexpected behaviour can be traced back to a wrong selection affecting one or more of these features.

Additionally, classical debuggers for imperative languages show values of variables in a straightforward way, since they are perceived as boxes holding their values, with no explicit relationship whatsoever with each other. However, values of variables in CLP are not simple items, but logical variables, and they can hold not only definite values, but ranges of values (either finite or infinite, depending on the domain of the language), and the basic elements in the domain can be integers, reals, boolean values, strings, etc., again depending on the domain. The actual range of a variable is updated as execution proceeds, being usually narrowed as the system advances towards a final solution, or being widened, when the system backtracks to search using an alternative path. These changes in the ranges of the variables are of utmost importance to understand an execution and the sources of a possible low performance.

Variables in CLP are also related among them by setting up constraints. From the viewpoint of the user, this is performed just by stating the equations in the source language. But internally, the representation and management of the constraints is very involved, and equivalent constraints differently expressed (for example, over-constraining some variables) can greatly affect the execution time. The possibility of visualising the constraint store and its evolution allows the programmer to perceive the relationship among the source code and the behaviour of the constraints.

Additional facilities which would be of great help are the possibility of projecting the constraint store at a given point on some variables (for example, on the variables of a clause). Moreover, current CLP systems have evolved to tackle large, real world problems, which has forced the introduction of complex, specialised constraints, such as the cumulative constraints [1.2], for which *ad hoc* representations are better suited. Visualisation of an instance of the cumulative constraint should show its evolving profile at runtime.

In any of the above cases, large executions pose a general problem: the amount of information to be treated or displayed cannot be dealt with using standard techniques, and special means of *abstraction* have to be applied. The aim is to allow the user to focus on interesting properties of the program, and to abstract automatically from irrelevant details.

In the next sections we will show how the above mentioned needs are addressed by the tools developed in the project. Some of the tools are specialised for a particular CLP system, while others exemplify more general techniques amenable to be adapted to a variety of systems.

4.1 Showing Control Features

Although one of the basic and very interesting properties of constraint programs is that it can generally be understood declaratively, i.e., without looking at control aspects, a concrete control strategy which drives the execution flow (and, thus, the search) implicitly exists in the evaluation engine. Sometimes a lack of understanding of how this control operates makes a program difficult to debug from the performance point of view. This control has two important components: the search determined by the program and the search in labelling. The control in the propagation of constraints, although permitted by some systems, does not fall in the same category as the flow control. While it is true that different propagation schemes affect the runtime constraints and, therefore, may ultimately change the *shape* and performance of the execution, the relationship between the settings of the constraint propagation control and the shape of the search tree (or, more precisely, the labelling tree) are too distant to be considered intimately related and explored in the same way.

The Box Model and its Appropriateness for CLP. Classical Prolog debuggers are generally based on the well-known "box model" [1.3]. This is an execution model where a procedure invocation is seen as a box, with unidirectional input- and output gates, named *ports*. These ports are associated with events happening to this call/box: entering the box for first time (i.e., calling the predicate), exiting with success, reentering to try another choice (i.e., redoing), and exiting with failure. Other ports can be defined—and turn out to be necessary to represent the operational behaviour of CLP systems. For example, suspended constraints waiting for instantiation, if they are to be seen as regular goals, need a sort of *suspend* and *awake* port. This complicates the representation of the execution, since the control and constraint solving parts of the system are mixed in a single execution flow. This approach possibly complicates a text-based debugger too much, even if the classical commands to break the execution at some point, and to skip or dive into parts of the program are available. Further enhancements, such as commands to display the state of the stacks or to locate the current execution point in the overall picture are an important aid, but still the problem of big executions and representation of variables (which is a relatively simple projection in Prolog) and constraints remain to be solved.

It is possible to make use of graphics in order to represent the boxes more intuitively; a first approach would be to literally display them, ones inside the others, and provide a zoom in/out facility to navigate in them. However, the depiction becomes very complicated when there are more levels of nesting, and probably this representation is only of pedagogical interest.

Showing Control as a Tree Another well-known idea is to visualise the resolution tree. This representation lends itself to a variety of further refinements and additions which makes it quite appealing, despite its initial simplicity.

One of the strong points of a graphical depiction of control is that the "big picture" of the execution is unveiled, so that the programmer can look at the execution globally, both in the programmed search and in the labelling part of finite domain solvers. The programmed search, despite being the core search technique in Prolog programs, is, in general, second to the labelling procedures in CLP. This labelling can, in fact, be seen as a specialised, data-driven (often including non-trivial heuristics) search. Moreover, it is not fixed, since several constraint logic programming systems (as CHIP or Prolog IV), allow the programmer to provide parameters to the labelling predicate which tailor its behaviour to the application at hand. If a visualisation of the enumeration is available, the user can control at each node how the chosen strategy behaves. This, combined with visualisation of the range of variables, will provide the programmer with a good view of how the search space (as determined by the domain of the variables at each execution point) is being reduced.

Thus, it appears useful to develop tools which show globally how the execution proceeded, both from the point of view of the program search and of the labelling steps. Since it is interesting to show which variables are involved in a given node (predicate call), it appears advantageous to interface tools aimed at this representation with the control view. We will talk about these tools in the next sections.

In view of the above, a feasible general strategy to implement control-related depictions, and to interface them with other visualisation tools, is the following:

A generic search tree visualiser should be able to show graphically the nodes of both the programmed search and the labelling search trees.

- In the programmed search view, a node should be shown per predicate call, including predicate names and, if possible, depiction of the source code and of the runtime data (i.e., the values of the variables). For some domains, as the Herbrand domain, this can be taken care of directly by the tree visualiser, but as the complexity of the domains increases, it is probably a better design choice to interface the raw tree visualiser with other tools aimed at the visualisation of variables. Several control actions should be possible while displaying this tree, namely: stopping the execution, stepping forward and backward, and abstracting parts of the execution in order to avoid overwhelming the programmer with too many details (see below). In addition, it should be possible to obtain selectively (e.g. by clicking on the nodes of the tree) a (possibly graphical) view of variables and of the constraint store at the execution points corresponding to the nodes of the tree.
- For the labelling search view, the nodes of the tree correspond to the choices performed within the labelling procedure. This, combined with a variable domain visualisation, which highlights how the current domain of the variables is being narrowed, provide the programmer with a good view of how

the search space is reduced. Abstractions techniques can be applied, in a similar way to the programmed search view.

4.2 Showing Values of Variables

The values of variables drive the execution, and obtaining them is its ultimate objective. Knowing them at runtime is a comparatively easy task in the case of traditional languages, but in the realm of constraint languages the situation becomes more involved: definite values are now transformed into constraints which relate the values of the variables and which restrict the possible range of values a variable can take.

In principle all the variables in the program could be visualised (i.e., the store itself could be shown as a collection of variables), but it is clear that in most cases this would overwhelm the programmer with an unwanted amount of information. A possibility which reduces the number of variables to take into account is to select those variables which are reachable from a given program point; this can be done by hand (inspecting the program) or by using special tools. Alternatively, some variables can be marked especially in the source program with annotations which do not change the meaning of the program, but which are understood by programs aimed at debugging.

Depicting the variables themselves needs a way of representing their domains. While for some constraint domains no satisfactory solution has been developed so far (e.g., for linear constraints), for other, such as finite domains, reasonably easy to understand depictions can be used. Textual representation of the domains, or graphical depictions such as that developed by Micha Meier [1.9] for the Eclipse system [1.7] will be used in the tools described in the next chapters. In particular, the graphical representation in [1.9], based on assigning a dot (c.f., square) to each possible value of a variable, which is therefore shown as a collection of dots or squares, is at the same time compact and amenable to be used to represent the relationships among variables and its history in time.

4.3 Showing Constraints

Constraint debugging refers to the process of debugging programs by examining the constraint store, as opposed to examining the structure (and the execution) of the program. A constraint-oriented view is helpful both in correctness debugging, since by inspecting the store the programmer may detect wrong or missing constraints, and in performance debugging, because the structure of the constraints determines the propagation of updates inside the solver, hence the number of internal operations performed. A more in-depth discussion of this issue can be found in several related chapters.

The variable visualisation tools mentioned above do not give any direct insight into the relationships or mutual influence among variables. As the values of variables are updated by adding constraints (which restrict the domain of the variables, or relate different variables), exploring which constraints are active at a given point, and which constraints were used to perform propagation at some point, is also very interesting. In general, it is useful to show some or all the constraints in which a subset of variables are involved (thus projecting the store over these variables). Obvious constraint representations include text-based displays of (projections of) constraints using the source language, but, as it happened with values of variables, this may not always be appropriate if too many constraints or variables are involved. Additionally, a source-based representation of constraints is usually very difficult to understand intuitively, especially when the number of constraints grows beyond some threshold.

It is possible to develop graphical representations of the relationships among variables: an appealing possibility is to allow the user to interactively change values of variables, and see what are the effects of these changes on the other variables. Interestingly, this approach is orthogonal to the way in which variable values are depicted. This animated depiction gives an intuitive understanding of the way the variables relate to each other, and, thus, of the constraints placed on them. A static version of this representation can be built, as a 2-D grid in which the points which belong to the domains of two given variables are highlighted. This would show the combined effect of all the constraints on the two selected variables.

A hindrance to constraint debugging is the non-hierarchical, plain composition of the constraint store. Unlike programs and data, which are structured in many ways (procedures, predicates and rules, objects, tree structures, etc.), it is generally admitted that the store is a mere flat and huge collection of formulas with no structure whatsoever. Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (e.g., in Prolog IV). However, the programmer has usually in mind a structured vision of the program and the relationships of the elements in it. Understanding how the store relates to this vision is a very interesting task which we will look at more deeply below.

4.4 Abstracting

In executions of large programs, the debugging process has to cope with a sizeable number of calls, and with a large number of variables, having each many possible values, and related through many constraints. The programmer can be easily overwhelmed by the amount of information, so that no conclusion can be drawn directly from it. Thus, it is highly desirable to have methods which help to deal with these very important cases. Abstraction methods, amenable to be applied to as many cases as possible, would give the programmer a more friendly interface, by removing unneeded details.

In the case of control depiction as a search tree a clear possibility is to abstract parts of the tree, maybe by collapsing them [1.11]. The user might select which parts of the tree can be collapsed (perhaps those which correspond to parts of the program known to be correct), using either interactive graphical interface or assertions added to the program. It is also important not to loose important information when performing this abstraction: as an example, abstracting a tree by collapsing parts of it may cause data regarding the size of the tree not to be displayed. This data can alternatively be encoded as a tag in the collapsed tree, as a number or as a colour code. In general, many information items we want to retain, but which cannot be easily kept when abstracting, can be added with tags to the abstracted picture.

A similar case appears when displaying variables. In the particular case of finite domains, variables having large domains would need (if the aforementioned point-per-value representation is used) pictures with too many details in them. This would clutter the display, so that it would be difficult to draw any conclusion from the depiction. Also, quite often not all the values in the domain are of interest: if we are looking at how fast variables have their domains narrowed, it is usually of little interest which values are removed at each step. On the other hand, when we are more interested in knowing properties about the correctness of the domains of the variables, probably some points outside the current domain are not of interest (because we know they have been correctly removed), and some values currently in the domain are of interest only if they are removed (because we foresee they should be part of a final solution).

Some techniques, as domain compaction, which restricts which parts of a domain are to be visualised, can be devised and applied either by hand or sometimes in a semi-automatic fashion, maybe driven by annotations written by the user in the program source, or by pragmas automatically generated by analysers.

Too many variables to be visualised also call for abstraction of the display. When the domain visualisation tools are linked with tree visualisation tools, a natural option is to display only the variables in the node being studied. This is not always possible: the (very interesting) case of global constraints, in which many variables are related by a complex relationship, usually needs tailored visualisation, because it is not easy to reduce its solving process to a CLP-type search without implicitly dealing with the details of the solving algorithm (Chapter 12).

4.5 Controlling size and complexity of the store (S-boxes)

As mentioned above, no reasoning can be done on a flat representation of the store. This can be addressed by allowing the programmer to *structure the store by modifying the constraint granularity.* This would be achievable by structuring the store as a hierarchy of sub-stores, organised themselves in sub-stores etc. and giving the programmer the necessary tools to. Thus, the store is to be organised as an hierarchy of sub-stores. The tools supporting this idea should make it possible:

– to create and to modify the hierarchy,
– to focus on a local view of selected sub-stores,
– to navigate in two directions (search tree and propagation process) in these sub-stores.

Taking advantage of properties of the constraint narrowing operators and of the main algorithm computing the overall domain reductions, it is then possible to generate arbitrary levels of abstractions of the store by considering subsets of the store as global constraints (in order to simplify the representation).

A tool for store inspection has been designed and is comprehensively described in Chapter 11. It allows the user to add structure information to the source code by *marking selected goals in selected rule bodies.* The set of "interesting" constraints (e.g., finite domain and/or interval constraints) defined by the corresponding predicates would then form a sub-store. Hierarchical organisation of the store is then associated to the program structure.

The key idea of the tool is to consider a store as a closed box (referred as *S-box* in the following) with no connection with the outside. By default, the store at any node of the search tree is considered as one S-box including all constraints of the store. This flat structure can be changed by using the tools to obtain an hierarchy of S-boxes, each of which includes a subset of the constraints involved. S-boxes can be created by highlighting some parts of the CLP program. This has the following effect, depending on what is highlighted:

– some constraints: they are all placed in one S-box;
– a goal: all constraints created when resolving the goal are included in one S-box;
– the head of a clause: all constraints created while in the clause are included in one S-box.

From the propagation view-point, a S-box acts as a big constraint which is the conjunction of all constraints it embeds. Note that this implies a modification of the propagation process: all constraints pertaining to the same S-box must be awaken in such a way that the whole propagation appears as atomic from the outside.

The set of constraints of the active S-box is displayed as a graph whose nodes are constraints and edges link constraints sharing variables. The graph contains only user constraints, that is decomposed constraints are reconstructed in order to be displayed. This requires modification in the core of the host system.

The hierarchy of existing S-boxes is represented in a tree form allowing the user to step backward and forward in the "zooming process".

The S-box approach can be seen as extracting constraints from the main CLP program, then gathering them in clauses to form a new structured Prolog program. The added structure facilitates debugging. In particular, it may allow more precise location of errors by the declarative debugger described in Chapter 5.

5 A unified view of the tools

This Section shows how the above mentioned tools relate to the proposed methodology and to each other. Figure 2 preserves the structure of Figure 1

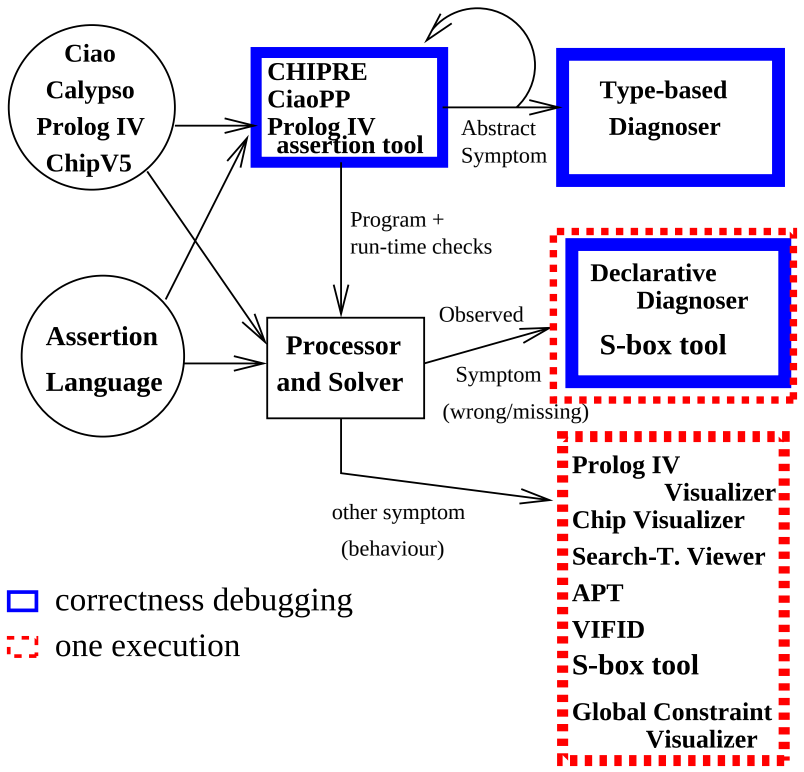

Fig. 2. Tools developed in DiSCiPl

to illustrate how the proposed methodology is supported by the tools. Most of the tools have been developed for selected individual platforms, as summarised below. However they follow the common view adopted in DiSCiPl.

As indicated in Figure 2, the platforms which have been used in the project are: Chip V5 [1.4], Prolog IV [1.10], Ciao [1.8], and `clp(fd)`/Calypso [1.6].

While the tools are platform-specific, the ideas represented by each of them are of general importance and can be integrated in any CLP platform.

In some cases porting of the tools does not require a big effort. This was demonstrated by porting the type analyser used in the type diagnoser to Ciao, integrating it in CHIPRE, and by porting the type analyser and type diagnoser to Calypso. Also, the assertion language (see Chapter 1) has been used for different tools and on different platforms.

The rectangles with thick edges in the figure indicate tools for correctness debugging. They include the tools for static debugging:

- **CHIPRE** and **CiaoPP** (instances of the **Generic Preprocessor**) (Chapter 2) use abstract interpretation to infer assertions about the program at hand. The inferred assertions are compared with the specification (check assertions), which may or may not be provided by the user, and with system assertions describing library predicates. If they do not conform the user is warned. For the check assertions that cannot be proved nor disproved run-time tests can be inserted in the program. Erroneous constructs are located by means of disproved check assertions and are reported to the user.
- The **Prolog IV assertion tool** (Chapter 3) verifies check assertions written in a restricted language, which must be provided a priori, and reports errors located by means of disproved assertions. Run-time tests can be included for assertions that cannot be proved nor disproved by the tool.
- The **Type-based Diagnoser** (for CHIP and for Calypso)(Chapter 4) uses the static analyser for inferring types. Diagnosis can be requested when one of the inferred types is different from that expected by the user (*abstract symptom*). Check assertions necessary for locating the error are requested interactively until an error message is obtained.

The rectangles with split Ted edges indicate tools for debugging based on run-time symptoms.

The **Declarative Diagnoser** implemented for Calypso (Chapter 5) is an interactive tool that locates errors causing symptoms of wrong or missing answers.

A lot of effort has been devoted to performance analysis, since it is probably the most difficult aspect of constraint debugging. The tools for performance analysis are visualisation tools which facilitate understanding of a single execution of the debugged program. As discussed above, the problems of particular interest are: visualisation of the search space, display of variable bindings and constraints, and analysis of the behaviour of constraints.

As mentioned before, the whole search space during a single execution of a CLP program can be represented by a tree structure. Several equivalent presentations of this structure are possible. In CLP programs over finite domains the performance depends heavily on labelling. Therefore, it may be desirable to have a separate visualisation tool only for labelling. These considerations are behind the design decisions for the DiSCiPl visualisation tools for search space. The following tools have been developed for this purpose:

– **Prolog IV Search-Tree Visualiser** (Chapter 6) presents the search space as a three-dimensional dynamic tree based on the trace box model.
– **CHIP Labelling Visualiser**(Chapter 7) presents the labelling part of the search space of an execution of a finite domain CHIP program.
– **INRIA Search-Tree Visualiser** (Chapter 8) visualises the traversed search space (including labelling) as an SLD tree and includes abstraction functionalities. The visualiser has been developed for Calypso. A special focus is on view abstraction, so that the same search space can be represented in different ways. Properties defined with assertions are used for that purpose.
– The **APT** visualiser (Chapter 9) uses and/or trees for representing the search space. It has been developed for Ciao and was also ported to Calypso.

In all the above mentioned tools the nodes of the search space visualised give access to data stored in the corresponding state of the computation. However, this may not be sufficient to understand the behaviour of constraints during the computations. The latter problem is addressed by the following tools:

– **VIFID/TRIFID** (Chapter 10) is a tool to visualise the data evolution of CLP(FD) programs in Ciao. It depicts in an intuitive way the state of the constraint store at selected points in the program. It gives the user several facilities, including allowing to post/un-post arbitrary constraints during the execution of the program. It also shows the evolution of (a subset of) the variables in the program at different levels of abstraction.
– The **S-Box** tool (Chapter 11) for `clp(fd)` facilitates analysis of constraint propagation by allowing the user to impose some temporary structure on the constraint store. It may be used both for performance debugging and for correctness debugging as a support tool for the declarative diagnoser.
– **Global Constraint Visualiser** (Chapter 12) for CHIP makes it possible to visualise the global constraints `cumulative, diffn, cycle` and `among`. This is of great practical importance since the global constraints are heavily used in all industrial applications and due to their sophisticated nature it may be very difficult to understand their behaviour.

Some of the tools discussed in the book have been already used in industrial applications. Chapter 13 discusses this experience and draws some conclusions from it.

1

1.1 E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1982.

References

1.1 A. Aggoun, F. Benhamou, F. Bueno, M. Carro, P. Deransart, W. Drabent, G. Ferrand, F. Goualard, M. Hermenegildo, C. Lai, J. Lloyd, J. Małuszyński, G. Puebla and A. Tessier. *CP Debugging Tools.* Public Deliverable D.WP1.1.M1.1, ESPRIT IV Project DiSCiPl, http://discipl.inria.fr/deliverables1.html, 1997.

1.2 N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1993.

1.3 L. Byrd. Understanding the control flow of prolog programs. In S.-A. Tärnlund, editor, *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

1.4 Cosytec SA. *CHIP System Documentation*, 1998.

1.5 P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

1.6 D. Diaz. *GNU-Prolog user Manual.* http://pauillac.inria.fr/ diaz/gnu-prolog/, 2000

1.7 European Computer Research Center. *Eclipse User's Guide*, 1993.

1.8 M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming.* Nova Science, Commack, NY, USA, April 1999.

1.9 M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`

1.10 PrologIA. *Prolog IV Manual*, 1996.

1.11 C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In Lee naish, editor, *ICLP'97.* MIT Press, July 1997.

1.12 E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1982.