# Practical Investigation of Constraints with Graph Views

**Tobias Müller**
Programming Systems Lab, Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
Email: `tmueller@ps.uni-sb.de`

**Abstract**

Combinatorial problems can be efficiently tackled with constraint programming systems. The main tasks of the development of a constraint-based application are modeling the problem at hand and subsequently implementing that model. Typically, erroneous behavior of a constraint-based application is caused by either the model or the implementation (or both of them). Current constraint programming systems provide limited debugging support for modeling and implementing a problem.

This paper proposes the Constraint Investigator, an interactive tool for debugging the model and the implementation of a constraint-based application. In particular, the Investigator is targeted at problems like wrong, void, or partial solutions. A graph metaphor is used to reflect the constraints in the solver and to present them to the user. The paper shows that this metaphor is intuitive and that it scales up to real-life problem sizes.

The Constraint Investigator has been implemented in Mozart Oz. It complements other constraint debugging tools as an interactive search tree visualizer, forming the base for an integrated constraint debugging environment.

**Keywords:** Constraint programming, correctness debugging of constraints, visualization of constraints, program analysis tools.

## 1 Introduction

The state of the art of solvers based on constraint propagation has made tremendous progress [5, 15, 18, 14], to the point where large combinatorial problems can be tackled successfully. But developing such applications has only limited support by debugging tools. This deficiency has been identified and dedicated projects (as DiSCiPl [7]) have been set up.

The first step to be taken when solving a combinatorial problem is to design a constraint model of the respective problem, i.e., to find a problem formulation in terms of constraints. Next this model is implemented by some constraint solver. Testing the implementation reveals quite frequently that no solution can be found, the solution found is not correct, or the solution found still contains undetermined variables. These situations suggest that the constraint model or its implementation do not reflect the combinatorial problem to be solved. To

In Konstantinos Saonas and Paul Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, Paris, France, 27–28 September 1999.

support the development process at this stage, the programmer needs adequate interactive debugging tools which are currently not available.

Current constraint debugging tools focus on improving search behavior [16, 2, 10], i.e., on finding search heuristics[1] for exploring the search tree most efficiently. There is a lack of intuitive interactive tools for debugging the correctness of constraint models and/or their implementations. In particular, large problems need tools with a sophisticated presentation to handle the overwhelming amount of information. Hence, providing an appropriate metaphor to present the data is crucial. The model of data presentation proposed in this paper is derived from graph-based visualization, as proposed by Carro and Hermengildo in [3]. The graph metaphor was first formally introduced in constraint programming by Montanari and Rossi [11].

The contribution of this work is the development of different graph-based views for correctness debugging constraint programs and the proposal of debugging methodologies based on these views for frequently occurring incorrect behavior of constraint programs. Furthermore, the techniques are extended to be able to handle large problems.

To prove the viability of our approach, we have designed and implemented an interactive tool, the Constraint Investigator, that allows the user to investigate the state of constraints and variables in a constraint solver by analyzing the corresponding graph views. The Investigator meets the following requirements:

- It can deal with large problems.
- It is not restricted to any specific constraint system.
- It provides intuitive data presentation and interaction, while affording detailed insights about the solver.
- It is fully configurable by the user.
- It requires no changes to the actual constraint program.
- It is suitable for users at different levels of expertise.
- It reveals operational aspects of the solver by displaying the events that trigger constraints.

The Constraint Investigator is implemented in Mozart Oz [12] and the visualization of the graph views relies on *daVinci* [17].

The Investigator complements the Oz Explorer [16], an interactive visual search engine, which does not take into account the aspect of constraint propagation. Both tools form the base of an integrated constraint debugging environment.

Although the Constraint Investigator is especially designed to tackle problems concerning the accuracy of the constraint model and its implementation, it can be easily extended for performance debugging. For example, its graph

[1]A search heuristics determines the policy of traversing the search space of a problem.

views can be used to present execution costs of constraints such that the program code causing these costs can be quickly identified. Furthermore, insights gained about the structure of a constraint graph enable the programmer to improve the (propagation) performance of the corresponding program. Especially since operational aspects of constraint execution (see Section 3 about events) are also presented to the user.

**Plan of the paper.** Issues of debugging constraint programs are discussed in Section 2. Section 3 introduces notions and concepts related to propagation-based constraint solving. The model of the Constraint Investigator is discussed in Section 4. The Investigator itself is explained by means of a prototypical debugging session in Section 5. Furthermore, Section 5 presents techniques for handling large problems. The implementation is sketched in Section 6. The paper closes with related work (Section 7) and concluding remarks (Section 8).

## 2 Debugging Constraints

Debugging an application focuses first on correctness and then on performance. Approaches to debugging can be identified as *experimental* and *analytic* [10]. Experimental debugging, i.e., modifying the program text until it seems to work, requires a large set of methods to experiment with. In contrast, analytic debugging needs to obtain a detailed description of the state of the constraint solver. Such a description has to be presented to the programmer by a debugging tool in a way that supports program analysis in the best possible fashion.

After designing and implementing the constraint model of a given problem, testing the implementation typically produces erroneous situations as:

- The solver fails immediately, i.e., the constraints are inconsistent. Either the implementation of the constraint model is incorrect or the model itself is. It is often the case that by accident the constraint model is over-constrained though the combinatorial problem is not. For example, the model states an equivalence where an implication is required. In such a case, if a solution is available (perhaps manually derived), it is a promising strategy to debug this situation by adding this solution to the constraint statements. The propagator which is observed to fail is not necessarily the culprit for the bug in the implementation but it helps to track down the problem in the constraint model.
- Propagation is incomplete in the sense that some solution variables remain undetermined. This is an indicator that the implementation or the model is incomplete.
- The solution found is wrong. Either the constraint model is incorrect or if this is not the case, the implementation of the model is incorrect.

The proposed debugging approach and the corresponding tool are aimed at analytic correctness debugging, i.e., to spot bugs in the constraint model and its implementation.

Analytic debugging requires an interactive tool that enables the programmer to analyze the actual constraints in the solver. The amount of information, i.e., typically the number of variables and constraints, is huge. The way these data are presented in analytic debugging is important since constraint programs are data-driven and an appropriate presentation helps the programmer to draw the right conclusions. Hence, data representation has to match the programmer's intuition of constraints in a constraint solver. Consequently, we choose a graph-based metaphor for representation since it makes possible to emphasize different aspects of the state of a constraint solver appropriately (see the different views presented in Section 4) and to relate the program structure to the representation (see Section 5.2).

## 3 A Model for Propagation-based Constraint Inference

Propagation-based constraint inference involves a *constraint store*, holding so-called *basic* and *non-basic* constraints. A basic constraint is of the form $x = v$ ($x$ is bound to a value $v$), $x = y$ ($x$ is equated to another variable $y$), or $x \in B$ ($x$ takes its value in $B$).

Non-basic constraints, as for example "$\neq$", are more expressive than basic constraints and hence, require more computational effort. A non-basic constraint is implemented by a *propagator* which is a concurrent computational agent observing the basic constraints of its *parameters* (which are variables in the constraint store). The purpose of a propagator is to infer new basic constraints for its parameters and add them to the store. A propagator terminates if it is inconsistent with the constraint store (`failed`) or if it is explicitly represented by the basic constraints in the store (`entailed`). A non-terminated propagator is either `sleeping` or `running`. A so-called *event* triggers the transition from `sleeping` to `running`. An event occurs when a basic constraint is added to the store. For example, a propagator might wait for a parameter to be bound to a value, while a different propagator has to be rerun as soon as an element is removed from a basic constraint connected to one of its parameters. A running propagator becomes either `sleeping`, `failed`, or `entailed`.

The constraints of a problem instance can be regarded as a network of propagators $P$, variables $V$, and events $E$. The variables in $V$ are the parameters of the propagators in $P$. The events in $E$ denote the changes to the basic constraints that trigger propagator transitions from `sleeping` to `running`. A propagator $p(v_1^{e_1 \in E_p}, \ldots, v_n^{e_n \in E_p})$ has a set of parameters $V_p = \{v_1, \ldots, v_n\} \subseteq V$ and is triggered by the events $E_p \subseteq E$. The notation $v_i^{e_i \in E_p}$ means that the propagator is rerun as soon as event $e_i$ occurs at parameter $v_i$. A variable $v(p_1^{e_1 \in E_v}, \ldots, p_m^{e_m \in E_v})$ is a parameter of the propagators $P_v = \{p_1, \ldots, p_m\} \subseteq P$ and changes to the basic constraint at $v$ can cause the events $E_v \subseteq E$. The notation $p_i^{e_i \in E_v}$ means that the propagator $p_i$ is rerun as soon as event $e_i$ occurs at the variable $v$.

## 4 Graph-based Visualization of Constraints

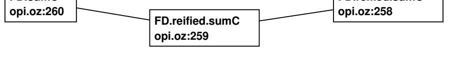
In this section, we illustrate different graph views using a trivial scheduling application. The problem is to serialize two tasks, such that they do not overlap. The first (second) task starts at starting time $T1$ ($T2$) and has a fixed duration of $D1$ ($D2$). The corresponding constraint model is the disjunction $T1 + D1 \leq T2 \vee T2 + D2 \leq T1$. The concrete implementation uses reified constraints to implement the disjunction. A reified constraint has an extra boolean parameter that reflects the validity of the constraint, i.e., whether it is `entailed` or `failed`. For example, $B1 = (T1 + D1 \leq T2)$ is the reified version of $T1+D1 \leq T2$ and if this constraint is `entailed` (`failed`) $B1$ is bound to $1$ ($0$). Conversely, in case $B1$ is bound to $1$ ($0$) the constraint $T1+D1 \leq T2$ ($T1+D1 > T2$) is stated. The (exclusive) disjunction of the constraints can be implemented by stating that the sum of the boolean variables associated with the reified constraints is $1$. The following Oz code implements the serialization constraint for two tasks:

```
B1 =: (T1 + D1 =<: T2)    % implemented by FD.reified.sumC
B2 =: (T2 + D2 =<: T1)    % implemented by FD.reified.sumC
B1 + B2 =: 1              % implemented by FD.sumC
```

Note that `D1` and `D2` refer to integers and all other variables are finite domains. The $=$-constraint is implemented by the Oz's finite domain operator `=:` and $\leq$-constraint by `=<:`.

In the following we present four different views of the above constraint program. The shape of a node represents its kind: a propagator node is a rectangle, a variable node an ellipse, and an event node a rhombus. A propagator node is annotated with the name of the respective propagator and the location of the propagator invocation in the source program, i.e., the file name and the line number. A variable node is annotated with the name of the respective variable and if the variable is constrained, the basic constraint connected to the variable is also shown. Note that there are no variable nodes for `D1` and `D2` since they denote integers.

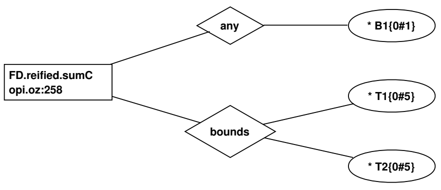
**The Propagator Graph View.** A propagator graph is the graphical representation of a propagator net, i.e., the propagators are the nodes. Note that the edges are not directed since data flow between propagators is bidirectional. This, for example, is different for a constraint solver using



indexicals [4] because an indexical is a function rather than a relation. For instance, the leftmost node corresponds to the propagator `FD.sumC` which happens to occur at line 260 of file `opi.oz` (the location of `FD.sumC` when we did the example graph views). This annotation depends on the concrete location of a propagator in a source file. An edge between two nodes means that the propagators share at least one variable parameter.

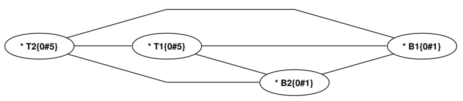
Using the sets $P$, $V$, and $E$ defined in Section 3, a propagator graph $pg(P_{pg})$ consists of nodes $N_{pg} = P_{pg}$ and edges $E_{pg} = \{(p_i, p_j) | V_i \cap V_j \neq \emptyset \wedge i < j\}$.

**The Single Propagator Graph View.** A single propagator view presents a single propagator and its parameters as a tree. The parameters are grouped by the events. Note a variable may occur several times as parameter. The single propagator graph view of `FD.reified.sumC` shows that the propagator waits for two events, namely the `bounds`-event, i.e., the bounds of the domain are narrowed, and the `any`-event, i.e., an arbitrary element is removed from the domain. Furthermore, the view shows that a `bounds` event at the parameters `T1` resp. `T2` and an `any` event at `B1` cause a rerun of the propagator. A variable node is annotated, as for example the node for `T1`: `*T1{0#5}` means that `T1` takes a value from $\{0, 1, 2, 3, 4, 5\}$. The asterisk ('`*`') denotes a variable passed directly by the user to the Investigator in contrast to variables collected while traversing the constraint network.



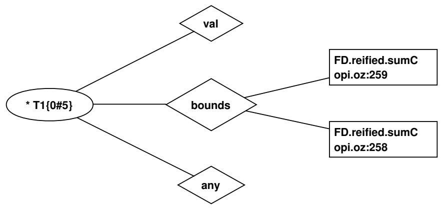
More formally, a single propagator graph $spg(p)$ for a propagator $p$ is a tree with a root node $R_{spg} = p$, connected to the root node are event nodes $E_{spg} = E_p$ and connected to the event nodes variable nodes $V_{spg} = V_p$. An edge between an event node and a variable node is established if the events of the event node and variable node are the same.

**The Variable Graph View.** A variable graph view is dual to the propagator graph view. The nodes represent the variables. An edge between two variable nodes indicates that the variables are simultaneously constrained by one or more propagators. The information of what propagators are concerned is available by a menu associated with the edge. The variable graph view shows that in our example, all variables are connected with each other.



The formal description of a variable graph makes the duality to a propagator graph obvious: a variable graph $vg(V_{vg})$ is composed by the nodes $N_{vg} = V_{vg}$ and the edges $E_{vg} = \{(v_i, v_j)|P_i \cap P_j \neq \emptyset \wedge i < j\}$. An edge between two variable nodes is present if the respective variables share at least one propagator.

**The Single Variable Graph View.** A single variable graph view represents a constrained variable, events it can cause and the propagators waiting for these events to happen. One can see that the two reified propagators wait for the `bounds` event and no propagator waits either for the `any` event nor for the `val` event.



A single variable graph $svg(v)$ of a variable $v$ is a tree with a root node $R_{svg} = v$. Event nodes $E_{svg} = E_v$ are connected to the root node. Furthermore, each event node of an event $e$ is connected to the propagator nodes $P^e_{svg} = \{p^e | p^e \in P_v\}$, i.e., an edge between an event node and a propagator node is established if the propagator waits for this event to happen to this variable.

## 5 Correctness Debugging with the Constraint Investigator

This section introduces the *Constraint Investigator* as an interactive tool for debugging practical constraint problems.

Using the Investigator does not require any changes to the constraint program. The program has to be recompiled with appropriate compiler switches.

### 5.1 An Example Session with the Investigator

We start off with a deliberately buggy constraint model and program and demonstrate how to track down two hidden bugs. Of course, the bugs are trivial to fix for experienced programmers but the approaches demonstrated are suitable for handling real-life situations.

**The Problem.** Consider the following bin-packing problem: a given set of weighted items $I$ has to be assigned to three bins $b_{1,2,3}$, without exceeding the maximum capacity of each bin. All bins have the same maximum capacity $c$. Furthermore, as soon as at least two items are put into a bin one extra unit of packaging material must be added as protection. Moreover, the bins must be color-coded to indicate the presence of a fragile item.

**The Constraint Model.** The given problem is a set partitioning problem of three sets with extra constraints. Each bin $b_n$ is modeled as set $s_n$ and each item $i \in I$ has a weight $w_i$.

$$
\begin{array}{llll}
I = \uplus s_n & (1) & |s_n| \geq 2 \rightarrow \text{packaging material } \in s_n & (2) \\
\Sigma_{\forall i \in s_n} w_i \leq c & (3) & i_{fragile} \in s_n \rightarrow \mathit{color}(s_n) = red & (4) \\
 & & \qquad\qquad \text{where} \quad n = 1,2,3 &
\end{array}
$$

Constraint (1) states a set partitioning and Constraint (2) adds extra packaging if necessary. Furthermore, Constraint (3) enforces that the capacity of the bins is not exceeded and takes also into account packaging material added by Constraint (2). The coloring of the bins is modeled by Constraint (4). The model is not quite correct as we will see later on.

**The Implementation of the Constraint Model.** The implementation of the presented model is based on finite set constraints [9, 13], i.e., a set value is approximated by a lower bound set and a upper bound set. The constraint solver has been implemented by the procedure `BinPacking`:

```
proc {BinPacking Weights Capacity Sol}
```

The argument `Weights` is a list of pairs `Id#Weight`. The variable `Capacity` determines the maximum capacity of the bins. The solution is returned in `Sol` and contains the colored bins with the assigned items.

The procedure starts with variable definitions: it declares the variables `Red` and `Green` for the bin-coloring constraint for the fragile item defined by `Fragile`. Next, it adds for the packaging material an extra item (`Packaging=100`) with weight $1$ to the list of all weighted items `AllWeights`. The list of `Items` is extracted from the weight list (`AllWeights`).

```
     Red     = 0    Green   = 1
     Fragile = 1    Packaging = 100
     WeightedPackaging = [Packaging#1]
     AllWeights        = {Append WeightedPackaging Weights}
     Items             = {Map AllWeights fun {$ E} E.1 end}
  in
```

The body of the procedure starts by creating the solution list `Sol` of length $3$. Each list element represents a bin as a record `bin(items:S color:C)` where `S` is the set of items and `C` is color of the bin. The application of `{FS.var.upperBound Items}` constrains `S` to the set constraint $\emptyset \subseteq$ `S` $\subseteq$ $setof($`Items`$)$.

```
     Sol = {List.make 3}
     {ForAll Sol
       fun {$}
          S = {FS.var.upperBound Items}
          C = {FD.int [Red Green]}
       in
          bin(items:  S
               color: C)
       end}
```

Next the partitioning constraint is stated (`FS.partition`). The `Map` function extracts the sets that form the partition from the bin records. The variable `Items` is converted to a set value by `FS.value.make` representing the set to be partitioned.

```
     % constraint (1): partitioning
     {FS.partition
      {Map Sol fun {$ S} S.items end}
      {FS.value.make Items}}
```

The weight restriction constraint maps the presence of elements to the list of boolean variables `BL` by `FS.reified.areIn`. The constraint `{FD.sumC ... ´=<:´ ...}` enforces that the scalar product of the list of boolean variables `BL` and the corresponding list of weights (produced by `Map`) does not exceed `Capacity`.

```
     % constraint (2): enforce weight restriction in bins
     {ForAll Sol
      proc {$ S} BL in
         {FS.reified.areIn Items S.items BL}
         {FD.sumC {Map AllWeights fun {$ E} E.2 end}
          BL ´=<:´ Capacity}
      end}
```

The constraints for adding packaging material and assigning the bin color close the procedure and use reified constraints. Reified propagators are used to conditionally state constraints according to constraint (3) in the constraint model. As soon as the cardinality of `S.items` is at least 2 the item `Packaging` is added to `S.items`. This is caused by the connection through the boolean variables of the reified constraints.

```
    % constraint (3): add extra packaging material
    {ForAll Sol
     proc {$ S}
        ({FS.card S.items} >=: 2) =:
        {FS.reified.include Packaging S.items}
     end}
```

The constraint for coloring the bins also uses reified constraints and implements the "$\rightarrow$" operator of constraint (4) by the implication constraint `FD.impl`[2].

```
    % constraint (4): assign colors to bins
    {ForAll Sol
     proc {$ B}
        {FD.impl
         {FS.reified.include Fragile B.items}
         (Red =: B.color)
         1}
     end}
  end % BinPacking
```

The code for controlling search is omitted since it is not of interest here and we assume an adequate search strategy. Now we submit our bin-packing solver to a search engine, like the Oz Explorer:

```
  {ExploreOne {BinPacking [1#3 2#2 3#2 4#6 5#2 6#4 7#3 8#5 ] 10}}
```

This results in an immediately failed search tree. The Investigator is now demonstrated in a prototypical debugging session.

**The Implementation is not Faithful to the Constraint Model.** Invoking the Investigator from the failed node switches the Investigator to the single propagator graph view (see Figure 1). The node representing the failed propagator is colored red throughout the session.

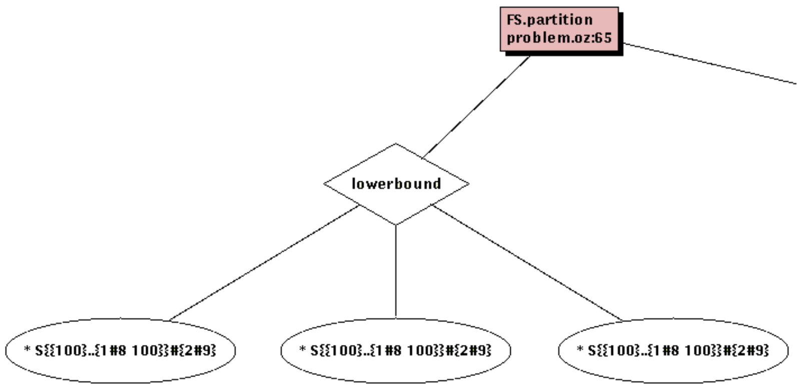
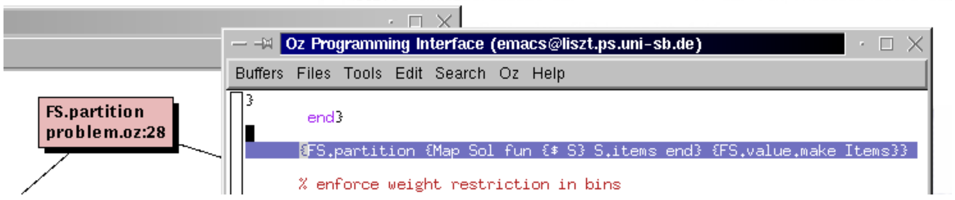

FS.partition
problem.oz:65
lowerbound
* S{{100}..{1#8 100}}#{2#9}
* S{{100}..{1#8 100}}#{2#9}
* S{{100}..{1#8 100}}#{2#9}

Figure 1: Single propagator view of the failed propagator `FS.partition`.

The single propagator graph view in Figure 1 shows the partition propagator with its parameters connected via the `lowerbound` event. The parameters are

[2]This is a reified constraint such that the last parameter `1` is required.

set constraint variables and are represented by `S{{100}..{1#8 100}}#{2#9}`[3]. This corresponds to the basic constraint $\{100\} \subseteq S \subseteq \{1,\ldots,8,100\} \wedge 2 \leq |S| \leq 9$. We notice that all three parameters contain at least element $100$. Hence, the partitioning propagator must fail. This reveals an incorrectness but this is not necessarily the actual bug. A single click on the propagator node highlights the line of source code where the partitioning propagator is stated (see Figure 2).


FS.partition
problem.oz:28
Oz Programming Interface (emacs@liszt.ps.uni-sb.de)
Buffers Files Tools Edit Search Oz Help

```
}
        end}

        {FS.partition {Map Sol fun {$ S} S.items end} {FS.value.make Items}}

        % enforce weight restriction in bins
```

Figure 2: Associating the failed propagator to the source program.

We see that the parameters concerned are the sets of items for each of the bins in the solution `Sol`. Checking the program text suggests that only the implementation of the packaging constraint (3) adds to all item fields of `Sol` the element `Packaging` (which is $100$). Verifying the code for adding extra packaging material reveals the bug in the implementation: instead of using different packaging material for each bin, the same material is used for all bins. This is not the intention of the constraint model and hence an implementation bug. The bug fix simply consists of using different packaging material items for each bin and modifies the `ForAll`–loop to select for different bins different packaging material.

```
% packaging material for every bin
WeightedPackaging = [(Packaging+1)#1 (Packaging+2)#1 (Packaging+3)#1]
...
{List.forAllInd Sol
 proc {$ I S} % 'I' counts from 1 to length of 'Sol'
    % select different packaging material by the index I
    ({FS.card S.items} >=: 2) =: {FS.reified.include 100+I S.items}
 end}
```

After fixing the implementation bug, we obtain as solution

```
Sol = [bin(color:0      items:{1#3 5 101}#5)
       bin(color:_{0#1} items:{4 7 102}#3)
       bin(color:_{0#1} items:{6 8 103}#3)]
```

and we notice that not all variables are bound to a single value (observe the `color` fields). The next section demonstrates how to track down the reason for this problem.

**Identification of Remaining Propagators.** A solution with unbound variables suggests that there is a lack of propagation. The variable graph view shown in Figure 3 is produced when starting the Investigator from the solution node of the Explorer.

[3]That all variables have the same name S does not mean that they are equal. The name is derived from the source code of constraint (1), cf. `{FS.partition {Map Sol fun {$ S} S.items end} ...}`.

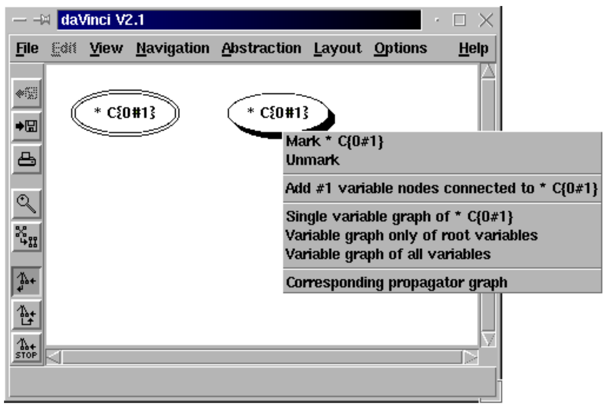
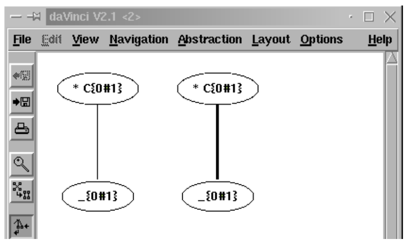
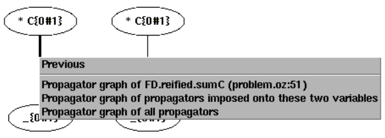

daVinci V2.1
File Edit View Navigation Abstraction Layout Options Help
* C{0#1}
* C{0#1}
Mark * C{0#1}
Unmark
Add #1 variable nodes connected to * C{0#1}
Single variable graph of * C{0#1}
Variable graph only of root variables
Variable graph of all variables
Corresponding propagator graph

Figure 3: Initial view.

The variable `Sol` is not displayed because it is bound to the solution list and hence no variable anymore. We try to find remaining propagators starting from one of the variable nodes. We decide to switch to the variable graph view of all reachable variables (Figure 4(a)), to get an overview over all variables left unbound. The menu associated with an edge between two variable nodes (Figure 4(b)) offers to switch to a single propagator graph view of a propagator being imposed upon two variables.



(a) Variable graph view of all reachable variables.

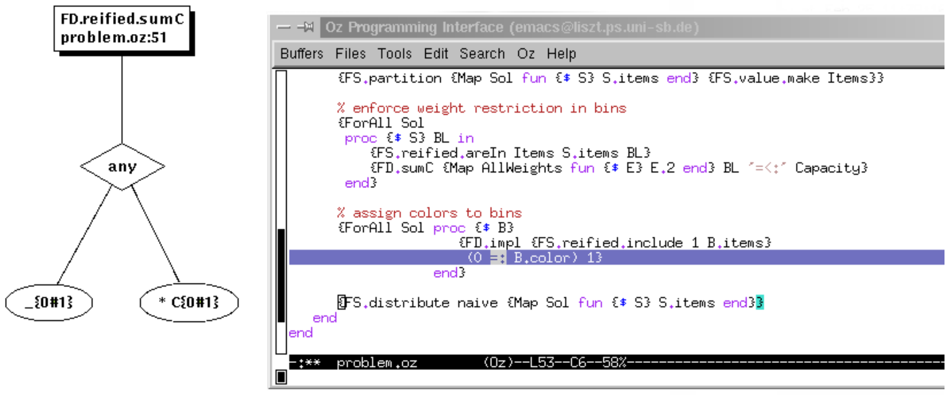

* C{0#1}
* C{0#1}
Previous
Propagator graph of FD.reified.sumC (problem.oz:51)
Propagator graph of propagators imposed onto these two variables
Propagator graph of all propagators

(b) Edge menu of the variable graph view.

Figure 4: Variable graph view.

Since we try to find remaining propagators, we switch to the offered single variable graph view of a reified sum propagator (Figure 5).

A click on the propagator node immediately reveals the suspicious program text: the assignment of the bin colors seems to be too weak whenever a fragile item is not contained in a bin (implementation of constraint (4)). The problem can be fixed by replacing the implication by an equivalence (`FD.equi`). The correct constraint (4) in the constraint model is $\forall n : i_{fragile} \in s_n \leftrightarrow color(s_n) = red$. That means that the implementation was correct but the constraint model had a flaw. After applying the fix the solver produces a proper solution.


FD.reified.sumC
problem.oz:51
any
_{0#1}
* C{0#1}
Oz Programming Interface (emacs@liszt.ps.uni-sb.de)
Buffers Files Tools Edit Search Oz Help

```
        {FS.partition {Map Sol fun {$ S} S.items end} {FS.value.make Items}}

        % enforce weight restriction in bins
        {ForAll Sol
         proc {$ S} BL in
            {FS.reified.areIn Items S.items BL}
            {FD.sumC {Map AllWeights fun {$ E} E.2 end} BL '=<:' Capacity}
         end}

        % assign colors to bins
        {ForAll Sol proc {$ B}
                          {FD.impl {FS.reified.include 1 B.items}
                           (0 =: B.color) 1}
                        end}

        {FS.distribute naive {Map Sol fun {$ S} S.items end}}
    end
end
-:**  problem.oz          (Oz)--L53--C6--58%----------------------------------------
```

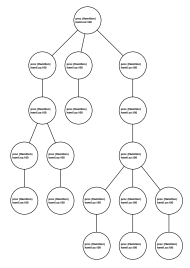
Figure 5: Single propagator graph view.

## 5.2 Dealing with Realistic Applications

Realistic problems may have thousands of propagators and variables. It is impossible and without any practical use to represent all at once. This section proposes techniques for selecting problem-relevant fractions of propagators or variables. This scheme allows for a user-controlled incremental exploration of the graphs which is essential for the investigation of large problems.

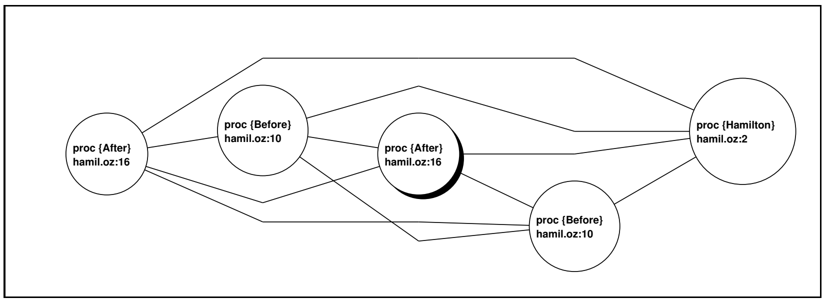
A common approach of designing a constraint model is to decompose the problem into subproblems and to decompose these subproblems until predefined propagators can be used. Since procedures implement subproblems, it seems reasonable to structure propagators, sub-procedures, and variables according to the procedures which stated them. This requires the introduction of procedure nodes to the graph views. A procedure node is depicted as circle.

**Selection via the Tree of Execution Traces.** The tree representation of a constraint program's execution trace (see Figure) is used to select propagators and variables. By clicking on a node, a possible action is to select the propagators created by the corresponding procedure invocation. Incremental expansion of the tree makes possible to handle large collections of propagators and variables. Different selection schemes, e.g., all propagators stated by a procedure with respectively without their sub-procedures, extend the functionality.

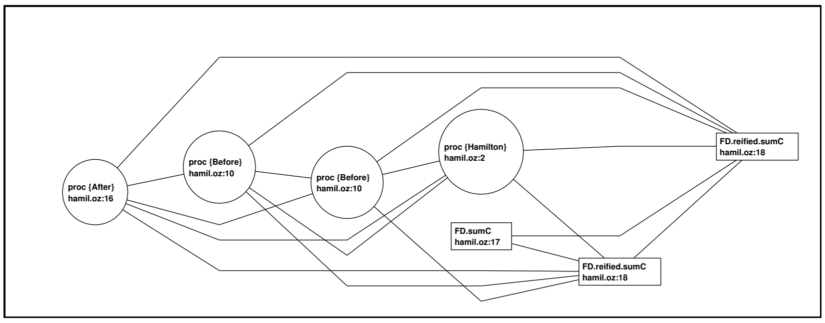


**Collapsing and Expanding Propagator and Procedure Nodes.** A common technique for handling large collections of data represented by graphs is to collapse and expand appropriate subsets of nodes to single nodes. We propose for the propagator graph view to determine subsets of nodes according to the procedures which created them. That means a collapsed node represents a collection of propagators and sub-procedures. This is very close to the model the programmer has in mind when structuring the problem and hence, is very intuitive.

A procedure node represents a collection of propagator nodes and sub-procedure nodes. It takes as its parameters the union of the parameters of all represented propagators and sub-procedures.



(a) Fully collapsed procedure graph, i.e., all propagator nodes are collapsed.



(b) Partially collapsed propagator graph, i.e., a procedures node is expanded to its propagator nodes.

Figure 6: Transition of a graph view by expanding a procedure node.

Figure 6 shows the expansion of the marked procedure node to a collection of propagator nodes. Expansion can be undone by collapsing propagator and procedure nodes to a single procedure node.

**Filtering propagators and variables.** Another interesting feature is the option of displaying only those propagators resp. variables which meet a criterion specified by the user. For example, it might be interesting to limit the investigation to those propagators that are connected to boolean variables when symptoms of a bug suggest that.

## 5.3 Additional Features

This section discusses features of the Investigator not covered before but important for effective use of the tool.

**Navigating through the Graphs.** Navigation through the different graph views is done by menus associated with nodes and edges of the respective views. Figure 7 shows possible transitions from one view to another one. A history mechanism is also available, allowing to recall previous views by moving in the chain of views produced so far.

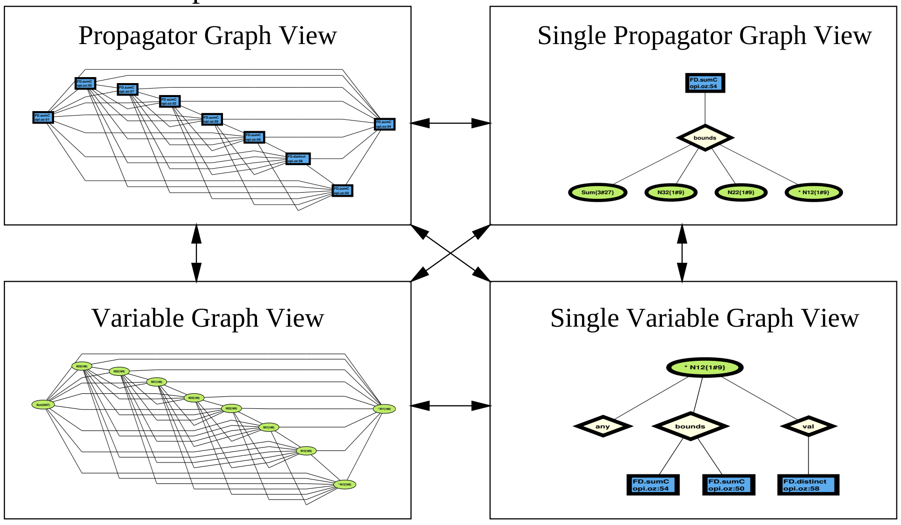
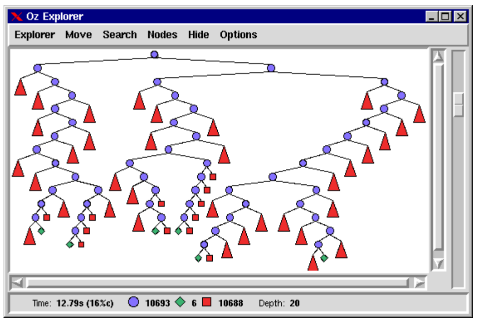


Figure 7: Navigation overview.

To further improve navigation and to keep track of a certain node in different views, the Investigator is able to mark nodes in graph views which then remain marked throughout all views.[4] Additionally, the Investigator automatically marks nodes of variables with which the session was initiated (Figure 3) and in case there is a failed propagator, the node of this propagator (Figure 1).

**Changing the Representation of Nodes.** The Investigator provides a plug-in mechanism for changing the representation of variables and propagators. This enables the user to produce a more obvious and intuitive representation. For example, propagators for cumulative constraints [1] in a scheduling application could be represented as Gantt-charts, reflecting their role in the concrete application.

**Interaction with the Oz Explorer.** The Investigator is designed to be integrated with the Oz Explorer [16]. The Explorer displays the search tree as search proceeds. The default action of clicking an Explorer node is to display the basic constraints of the corresponding constraint store. Plugging in the Investigator makes possible to investigate failed and unfailed search tree nodes by clicking on the respective nodes. The Investigator operates on Explorer nodes as demonstrated in the Section 5.1.



[4]For the purpose of this paper, a marked node is drawn with double lines. Other schemes, for example using colors, might be more suitable in connection with color displays.

## 6 Implementation

The Investigator is implemented in Mozart Oz [12] and uses the graph visualization system *daVinci* [17]. The implementation consists of two parts: first, the reflection of variables and propagators in the solver into Oz data structures, and second, the construction of the graphs and the generation of the corresponding *daVinci* terms.

The reflection part requires the Oz program to be able to have propagators as a first-class abstract data type, to be able to detect equal variables respectively propagators, and to have primitives to reflect variables and propagators to Oz data structures. The data structures make explicit the connections between propagators and variables, i.e., which are variables acting as parameters of propagators and which propagators are waiting for a variable's basic constraint to change. A data structure reflecting a variable stores the name, the basic constraint, the references to the propagators that are waiting for the variable's basic constraint to change, and the actual variable. A reflected propagator stores the propagator's name and parameters and a reference to the actual propagator. Reflection typically starts from the solution variables of a problem and proceeds by traversing all reachable propagators and variables until it terminates, i.e., when there are no more new propagators and variables to be found. Detecting termination, when implemented naïvely, needs to be able to test variables and propagators for equality. We have used a hash table here to overcome the quadratic complexity of the naïve implementation.

The list of newly found propagators and variables is translated into a compact representation for the connections between propagators and variables using sets of integers [13]. Each propagator and variable is assigned a unique integer. A reflected propagator stores its parameters as an integer set, also a reflected variable stores the propagators for which it is a parameter as an integer set. The edges and nodes of the individual graph views are computed by the set operations described in Section 4 and are further translated to *daVinci* terms. The *daVinci* terms are augmented by menus to allow for comfortable user interaction.

The complexity of the graph-generation algorithm is quadratic in the worst case and depends in practice on the degree of connectivity of the constraint network, i.e., if the propagators can be stated in reasonable time then the corresponding graph can be computed in reasonable time too.

Collapsing and expanding of procedures and propagators required extra data structures for procedures. These are an extension of the data structures used for propagators and contain additionally references to sub-procedures and propagators stated by the respective procedure. Due to the set-based implementation, the changes can be factorized out nicely.

## 7 Related Work

The tools discussed in this section focus on improving performance. Since our approach is orthogonal, it can be used to supplement existing tools.

The Grace constraint debugger by Meier [10] supplements the Prolog-based constraint programming system $ECL^{i}PS^{e}$ [8] and is intended to support performance debugging of finite domain constraint programming. The constraint program has to be appropriately instrumented to be run under Grace. The debugging model of Grace is based upon the Prolog-box-model. It is able to follow individual propagation steps in the trace and to inspect the backtrack stack of finite domain variables. Furthermore, Grace is highly configurable by assigning user-written code to each propagation step.

The Oz Explorer is a graphical search engine which visualizes the search tree as search proceeds. It allows the user to control search, e.g. one can interrupt search and can resume search from a branch different from the branch explored last. The Explorer is extendable by plug-ins, to provide different views of nodes in the search tree. The Explorer is particularly useful for optimizing search heuristics according to the topology of the search tree. In conjunction with the Investigator, debugging performance and correctness issues of constraint programs is actively supported.

The search tree debugger of Chip [2, 6] is largely influenced by the Oz Explorer[5]. Its focus is performance debugging. It provides different types of views, mostly in a compact matrix-like fashion, to provide the user with more detailed information about search and constraint propagation. A nice feature is to analyze the evolution of constraints and variables along a search path. This is certainly most valuable for optimizing search heuristics.

## 8 Conclusion

We have presented a novel approach for correctness debugging constraint programs based on graph views and have showed how this approach can deal with large constraint problems. Based on this approach, we have implemented an interactive tool, the Constraint Investigator. The use of the Investigator has been demonstrated with an example derived from a realistic constraint programming application.

The Investigator has been tested with problems of medium size (500 propagators and 600 variables) and has helped to understand and to debug the constraint-based implementation of a natural language parser.

To our knowledge, no other interactive constraint debugging tool uses a graph metaphor in the way presented in this paper and that makes the Constraint Investigator unique.

**Acknowledgements.** I would like to thank the *daVinci* team for making this brilliant tool available. I am grateful to Leif Kornstaedt for helping me with

[5]See in [2] Section 3 on related work.

the implementation. I am also grateful to Christian Schulte for many fruitful discussions. Denys Duchier exercised the Investigator in the development of linguistic applications and helped with invaluable comments and suggestions. The aforementioned, Katrin Erk and the anonymous referees gave invaluable comments on earlier versions of this paper.

The research reported in this paper has been supported by the Esprit Working Group CCL-II (EP 22457).

## References

[1] A. Aggound and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[2] A. Agoun and H. Simonis. Search tree visualization. Technical Report D.WP1.1.M1.1-2, COSYTEC SA, June 1997. In the ESPRIT LTR Project 22352 DiSCiPl.

[3] M. Carro and M. Hermenegildo. Some design issues in the visualization of constraint logic program execution. In *AGP'98*, A Coruña, Spain, 1998.

[4] P. Codognet and D. Diaz. Compiling constraints in `clp(FD)`. *Journal of Logic Programming*, 27(3):185–226, June 1996.

[5] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988. Institute for New Generation Computer Technology (ICOT),Tokyo, Japan.

[7] DiSCiPl. Debugging systems for constraint programming. `http://discipl.inria.fr/`.

[8] ECRC and International Computers Limited and IC-Parc. *ECL$^i$PS$^e$, User Manual Version 3.7*, February 1998.

[9] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[10] M. Meier. Debugging constraint programs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP95)*, 1995.

[11] U. Montanari and F. Rossi. True concurrency in concurrent constraint programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 694–713, San Diego, USA, June 1991. The MIT Press.

[12] The Mozart Consortium. *The Mozart Programming System*. `http://www.mozart-oz.org/`.

[13] T. Müller and M. Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.

[14] T. Müller and J. Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163. The MIT Press, Cambridge, 1997.

[15] Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Logic Programming – Proceedings of the 1995 International Symposium*, pages 513–527. The MIT Press, Cambridge, December 1995.

[16] C. Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 8-11 July 1997. The MIT Press, Cambridge.

[17] Universität Bremen, Group of Prof. Dr. Bernd Krieg-Brückner. *The Graph Visualization System daVinci*. `http://www.informatik.uni-bremen.de/~davinci/`.

[18] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECL$^{i}$PS$^{e}$: A platform for constraint logic programming. *ICL Systems Journal*, 12(1), May 1997.