

An Integrated Modelling, Debugging, and Visualisation Environment for G12

Andreas Bauer^{1,2}, Viorica Botea¹, Mark Brown¹, Matt Gray^{1,2},
Daniel Harabor^{1,2}, and John Slaney^{1,2}

¹ National ICT Australia (NICTA)*

² The Australian National University

Abstract. We present G12IDE, a front-end for the G12 platform aimed at helping users create and work with constraint models in a manner independent from any underlying solver. G12IDE contains tools for writing and evaluating models using Zinc and provides a feature rich debugger for monitoring a running search process. Debugging a search, as opposed to debugging sequential code, requires concepts such as breakpoints and queries to be applied at a higher level than in standard debuggers. Our solution is to let users define special events which, once reached in a search, cause the debugger to halt and give back, possibly in a visual manner, useful information on the current state of the search. G12IDE also includes a number of visualisation tools for drawing graphs and trees, and additionally allows users to create arbitrary domain-specific visualisations, such as the drawing of a sequential plan when the constraint problem is in fact a planning problem. The inclusion of such powerful and flexible visualisation toolkit and its tight integration with the available debugging facilities is, to the best of our knowledge, completely novel.

1 Introduction

G12 [15] is a software platform for solving combinatorial optimisation problems. It supports linear and mixed integer programming, constraint propagation and inference and a variety of other search and inference-based approaches for solving complex problems. Like several other modern modelling languages [5,6], it separates the “conceptual” or constraint model from the constraint program. In G12, the constraint model is written in Zinc, a purely declarative language that can be mapped to a range of lower-level models, and ultimately to constraint programs, which may solve the problem in quite different ways. The clear separation of modelling from solving requires a shift from thinking about problem solving in terms of programs and execution to thinking in terms of models and search. This change in paradigm calls for new tools that directly support working at such a high level.

In this paper we present G12IDE, a novel integrated modelling, debugging and visualisation environment which has been developed largely in parallel to the rest of G12.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Built on top of the Eclipse platform,¹ G12IDE has a similar look-and-feel to a classical programming environment. It offers support for the modelling of constraint problems via an integrated Zinc editor, performs automatic builds and allows step-wise debugging of an active search process.

Our debugging system is built around an explicitly defined (and solver agnostic) schema which specifies a wide range of “interesting” search events that a user may “subscribe” to. For example, it is possible to pause the search when a variable has become grounded, when the search has reached a fixpoint or when a solution has been found. Furthermore, existing search events can be combined together to create new custom search events. This allows for a high degree of control over the search process at varying levels of granularity.

Our environment also allows constraint problems to be visualised. Users can choose between pre-defined visualisations, such as a constraint graph and search tree, or alternatively may define their own. The latter is facilitated by drawing objects in a simple graphics editor and animating them using a dedicated scripting language. In this way, not only are generic views (trees, graphs) available, but it is also possible to visualise problems in a more domain-specific manner. For example, a planning problem can be visualised by drawing the execution of the plan so far as well as the possible choices in the current search node. The inclusion of such a powerful and flexible visualisation tool in the IDE, and its tight integration with the available debugging facilities, is to the best of our knowledge, completely novel.

Outline. The rest of this paper is organised as follows. In the next section, we provide more conceptual details on debugging a running search, outlining the differences from classical debugging and the specific challenges of our domain. In Sec. 3, we give a brief architectural overview on the G12IDE, and explain its main components, or layers. The modelling layer is explained in greater detail in Sec. 4, and the visualisation layer in Sec. 5. The technical realisation of our debugging layer, or rather its relation to the underlying constraint solver, is explained in Sec. 6. Finally, Sec. 7 contains references to related work, while a brief summary and conclusions of our paper are to be found in Sec. 8.

2 Debugging Search

Bugs—errors or infelicities—may exist anywhere in the software system. They may afflict the model, the data, the mapping down to solvers or the underlying programs. Bugs in the code of solvers are not our present concern: we assume the constraint program and its associated constraint solvers work perfectly unless forced to conclude otherwise. Instead, we focus on bugs that may arise during the development of a constraint model.

Errors in the model may call for *correctness debugging*, if they affect semantics by allowing unintended solutions or by excluding intended ones. It is also common for models to contain logically correct but poorly expressed constraints. Such situations can frequently limit the effects of propagation and so *performance debugging* of the model may also be required. We thus require debugging tools which detect either static

¹ <http://www.eclipse.org/>

features of constraint models or dynamic features of the search process, and report them to us in a form appropriate to the high level at which we wish to think about constraint problems.

In a sense, this is a radical departure, but in another sense it is little different from the concept of debugging at other levels. Consider a debugger for C programs, for instance. It works with an ontology appropriate to programming at that level. It allows breakpoints to be set on lines of the C code or on C functions, not on assembly-level instructions, and when stepping to the next instruction, it breaks on the next C line rather than on the next machine instruction or the next clock cycle. When it reports the values of variables, these are variables declared in C, not the contents of registers and accumulators. Moving up from the level of program and execution to that of model and search is more of the same. Breakpoints and steps for our purposes should make sense in terms of search: “step to the next node of the search tree” and “break whenever propagation reaches a fixpoint” belong at this level, whereas “step over the next function call in the SAT solver” does not. Similarly, when we ask for the values of variables, we want to know the domains of decision variables declared in the Zinc model, not the details of whatever data structures these have turned into after mapping to solvers.

The task of our debugger is to monitor the search process, which requires it to place breakpoints in low-level code in order to collect information with which it can maintain models of the current search state, and then to pass just the right information, on demand, to the front-end tools which display it. Managing this in a systematic way seems to be new in constraint programming, so we have had to design an architecture for the search debugger at the same time as experimenting with modes of visual presentation in order to present abstractions of the search states which are likely to be useful. Details of our design decisions and the resulting tools are presented in the next sections.

3 Architectural Overview

Our environment can be described in terms of a three-tier architecture comprising a Modelling Layer, a Solving Layer and a Visualisation Layer. Fig. 1 illustrates this idea.

The Modelling Layer is where most of the interaction with the user takes place. It comprises two components: The first is a dedicated code editor for Zinc which offers features such as syntax checking, syntax highlighting, and standard Eclipse functionality such as project-based code separation. The second component is a Visualisation Editor which includes a simple canvas for drawing objects and an associated text editor for writing animation scripts. The idea is to create a custom visualisation by drawing objects on the canvas and then writing a script to define their behaviour in response to specified search events (e.g. changing the colour or position of an object in response to a variable becoming grounded). Sec. 4 describes the components of the Modelling Layer in more detail.

The Solving Layer comprises the main interface to the rest of the G12 platform (i.e., the different solvers) and our environment. Given a Zinc model, the Solving Layer is responsible for invoking the solvers, maintaining communication between the solver monitor and the debugging interface and is responsible for sending any updates regarding the state of the running search process to the appropriate visualisers. Sec. 6 contains a detailed discussion of the different components in the Solving Layer.

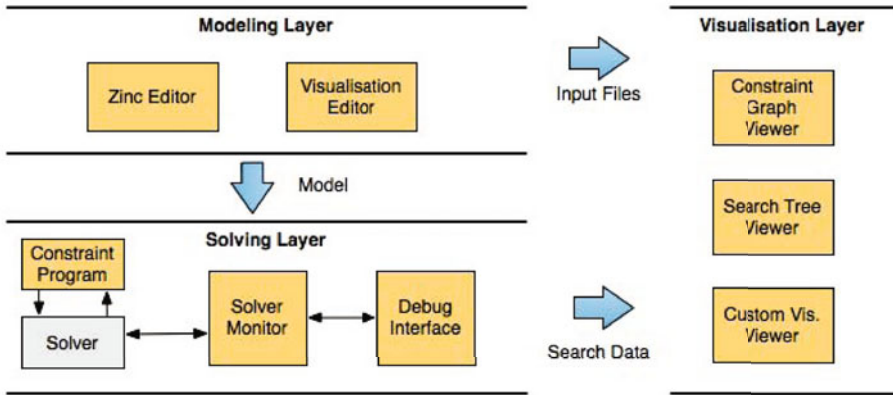


Fig. 1. The G12IDE architecture. Most features can be categorised into one of three distinct layers: modelling, solving or visualisation.

The Visualisation Layer is comprised of three dedicated visualisers: a Constraint Graph Viewer (CGV), a Search Tree Viewer (STV) and a Custom Visualisation Viewer (CVV). The first two are highly specialised, offering a range of features specific to the display of graphs and trees. The CVV on the other hand is extremely generic; while intended principally for 2-dimensional diagrams, it can display arbitrary graphics. Its applicability is thus limited only by the constraint programmer’s imagination. Each visualiser requires certain input files to function. In the case of CGV and STV the model itself suffices. The CVV also requires a visualisation script. For details on these viewers as well as the scripting language, see Sec. 5.

4 The Modelling Layer

Two components make up the Modelling Layer: a Zinc Editor and a Visualisation Editor. To better illustrate these tools and their use, we shall refer to the *meet-pass problem* as a running example. This is a standard (if not very difficult) benchmark problem in AI planning, described as follows:

The Meet-Pass Problem

Five sectors of railway track, S_1, \dots, S_5 , are linearly connected. There is a siding accessible from S_3 big enough to hold one train. Initially, there are trains in sectors S_1 , S_2 and S_4 . The safety rules are that no two trains may be in the same sector at the same time, and no train may enter a sector occupied by another train, even if that other train is about to move on. Trains may only move to adjacent sectors, of course. Find the shortest plan that moves the train on S_1 to S_5 and returns the other two trains to their starting positions.

While this is a toy example, the problem class from which it comes is real enough: meet-pass planning is a constant issue in scheduling train movements. Fig. 2 shows a

```

set of int: Sectors = 1..6;
set of int: Trains = 1..3;
set of int: Steps = 1..13;
array[Sectors,Sectors] of bool: linked;
array[Trains] of Sectors: start;
array[Trains] of Sectors: finish;
array[Steps,Trains] of var Sectors: pos :: is_output;

constraint forall(t1, t2 in Trains, x in Steps)
  (pos[x,t1] == pos[x,t2] -> t1 == t2);
constraint forall(t in Trains)
  (pos[1,t] == start[t] /\ pos[nsteps,t] == finish[t]);
constraint forall(t in Trains, x in Steps where x > 1)
  (linked[pos[x,t],pos[x-1,t]]);
constraint forall(t,u in Trains where t != u)
  (forall(x in Steps where x > 1)(pos[x-1,u] != pos[x,t]));

start = [ 1, 2, 4 ];
finish = [ 5, 2, 4 ];
linked = [| true,  true,  false, false, false, false
          | true,  true,  true,  false, false, false
          | false, true,  true,  true,  false, true
          | false, false, true,  true,  true,  false
          | false, false, false, true,  true,  false
          | false, false, true,  false, false, true |];

solve satisfy;

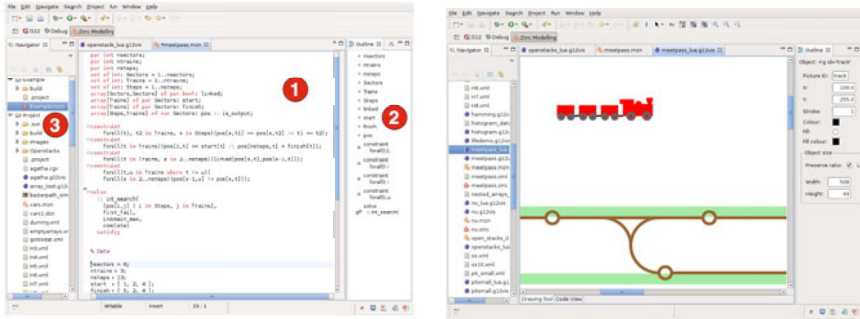
```

Fig. 2. Meet-pass planning problem in MiniZinc

MiniZinc encoding of the problem given that the optimal plan length is known to be 12 moves (13 timesteps). The four constraints are quite simple: the first is a safety condition which could be written using the `all_different` global if we wished; the second is trivial; the third says that trains can't jump and the fourth encodes the remaining safety condition.

4.1 The Zinc Editor

The Zinc Editor (shown in Fig. 3(a)) allows users to write constraint models in the main input languages of the G12 platform: Zinc and MiniZinc [12]. It has a range of features which are typical for code editors in other integrated development environments. For example, automatic syntax checking and highlighting, in-line error reporting, and an outline window to assist with code navigation are standard. Another largely standard feature offered by the Zinc Editor is project-based code management, which simplifies the task of keeping models, data files and related visualisation scripts together. Since the Zinc Editor directly extends the standard Eclipse code editor it is easy to augment its functionality via third party libraries or “plugins”. For example, it is trivial to add support for other programming languages (such as java) or add features to help with revision control.



(a) The Zinc Editor includes features common to many programming environments: syntax highlighting (1), outline views (2) and project-based code management (3) are all standard.

(b) The Visualisation Editor drawing tool. Shown are some domain-specific shapes (trains, tracks) that we have drawn for the Meet-pass problem.

Fig. 3. The two main built-in editors of the G12 IDE

4.2 The Visualisation Editor

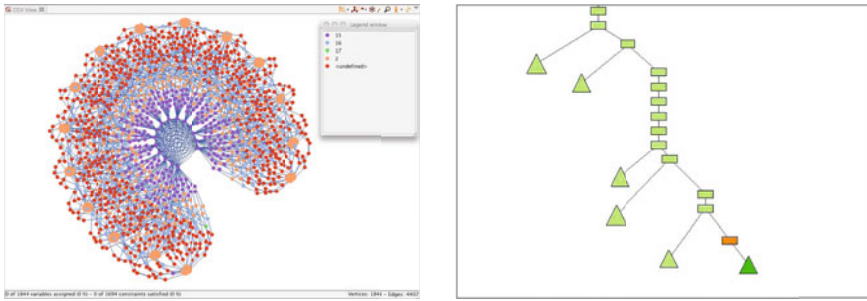
The Visualisation Editor is a vector-oriented drawing tool and an associated script editor which together are used to define arbitrary visualisations of the model as it is being solved.

A new visualisation is created by first drawing a small set of shapes (in the drawing tool) where each shape represents an object or concept specific to the domain of the constraint problem. In the case of Meet-pass for example we might draw a train and some different types of tracks (as shown in Fig. 3(b)). Next, a visualisation script is created which animates the drawn objects – usually in response to some search data received from the solving process. Our visualisation scripts are written in the Lua language and thus both flexible and powerful. We chose Lua because it is portable and lightweight but as we will discuss in Sec. 5, G12IDE can be extended to support any arbitrary programming language.

The basic operation of a visualisation script is straightforward: Each shape defined via the drawing tool is available as a template that can be instantiated by the script. Once an instance is created the shape can be programmatically positioned to anywhere on the canvas. Other attributes of the templated shapes (such as their size, colour, orientation and opacity) can be likewise modified at any point. Further details related to the integration of the scripting language into G12IDE and the operation of the Custom Visualisation Viewer are given in Sec. 5.

5 The Visualisation Layer

G12IDE offers three distinct visualisers: a Constraint Graph Viewer (CGV), a Search Tree Viewer (STV) and a Custom Visualisation Viewer (CVV). The first two are pre-defined and highly optimised visualisations specific to constraint problems and search.



(a) A typical constraint graph. Nodes represent variables and edges, constraints.

(b) A typical search tree. Each node is a domain split. The deltas are collapsed subtrees.

Fig. 4. The search tree and constraint graph visualisers

The third is more general; it can display any arbitrary visualisation which has been developed with the Visualisation Editor (see Sec. 4). We discuss each in turn.

5.1 Pre-defined Visualisations

The STV visualises a constraint solver exploring the search space. It expands and draws a search tree in a step-wise manner that shows decision points, backtracking operations, domains of variables and other information which is useful when inspecting a search tree. Fig. 4(b) shows a typical result. A range of controls is available to speed up, slow down, pause and resume the solving process. It is also possible to collapse and expand entire branches of the tree to speed up drawing time. Additionally, as tree search algorithms can take a long time to complete, we have implemented the “recursive tree estimator” of Kilby et al [10] to provide the user with some indication of how long the solving process is likely to take.

The CGV visualises the solving process using a constraint graph—a useful representation of the structure of a problem. In its simplest form a constraint graph is composed of a set of nodes representing decision variables and a set of edges which represent constraints between them. An alternative representation draws each constraint as a node and adds an edge only if two constraints share a decision variable. CGV supports both types. Updating CGV during the solving process usually amounts to either highlighting or graying out or even hiding instantiated variables and satisfied constraints, and then recovering them on backtracking. This allows the user to see the search progressing through the constraint graph or maybe jumping from one part of it to another. Adding and deleting constraints during the search is less common, but occurs for example when SAT solvers learn nogoods. As the number of constraints and variables in a typical model can be quite large a natural problem which arises is how best to draw the graph. We solve this issue by adopting various force-directed layout algorithms [13,7]. Fig. 4(a) shows a typical result for a scheduling problem. For small graphs, the user may prefer to position nodes by hand: CGV also provides a drag-and-drop facility to support this. The same set of controls used by the STV over the solving process also apply to CGV. In fact, both CGV and STV can be displayed at the same time.

5.2 Custom Visualisations

Both CGV and STV are problem-independent and always available. However, it is often the case that a constraint problem lends itself to a domain-specific visualisation which is more informative. This might show a partial plan for a planning problem, an unfinished Gantt Chart for a scheduling problem or, in the case of one well known CSP problem, queens on a chess board.

Accordingly, we have developed a Custom Visualisation Viewer (CVV) which allows users to create their own domain-specific visualisations in two steps: first, users draw some domain specific objects in the Visual Editor’s drawing tool; e.g., a train, a queen, and so forth. Second, users write a short script which controls how these shapes should be drawn on the canvas. To facilitate the creation of such scripts G12IDE offers a very simple API that allows for arbitrary scripting languages to be integrated and used. As a proof of concept we currently provide support for the Lua [8] language.

Any arbitrary visualisation script will, as a minimum, need to implement a `step()` method which will be invoked every time the debugger is paused as a result of reaching some break condition (see Sec. 6 for more details about break conditions). In the course of the `step()` function the script will usually need to call one or both of the following methods (possibly multiple times):

- `g12GetFromData(var)`. This method fetches the current contents of the decision variable `var` and makes its value available to the script.
- `g12Draw(obj, props)`. This method sets a list of properties, `props`, for object `obj`, where `props` may contain items such as the object’s positions on the canvas, colour, opacity, scaling factor, etc.

5.3 Support for Other Scripting Languages

The choice to use Lua as the default language of the CVV was a simple one: it is portable, lightweight and often used in similar contexts (for example by the video games industry). Lua also offers an intuitive syntax, straightforward control and data structures as well as a simple type system that can be picked up with minimal effort by anyone familiar with an existing programming language. However, we could as well have chosen Python, Lisp, or any other language which supports Java integration. If besides Lua, we wished to offer, say, Python as a scripting language, we would basically have to implement the above `g12-*` calls in a Python program that in turn invokes the corresponding methods of our IDE. There are only two files specific to the integration of Lua, whose length is less than 1000 loc: the first file is a Lua script which contains the API calls to the IDE, and the second a Java class handling the invocation of the Lua interpreter. How this interaction between the script, the underlying interpreter and the main IDE is technically realised is schematically depicted in Fig. 5. The aforementioned two files are depicted as “Lua Plugin”, written entirely in Java, and “Lua API”, written in Lua. Strictly speaking, the latter is part of the Lua Plugin itself and provided by it. The user who writes a Lua script, merely has to include this file to be able to issue the relevant `g12-*` calls from inside the Lua script.

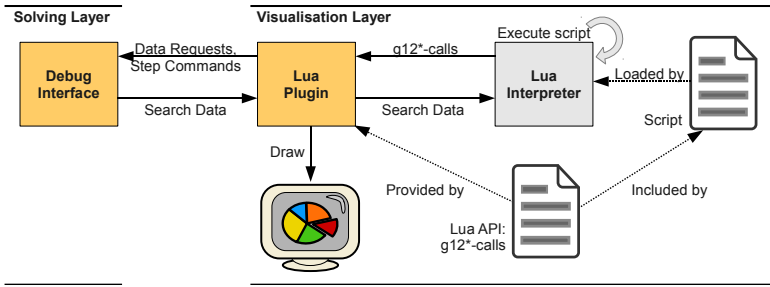


Fig. 5. Schematic overview of visualisation of user-defined scripts. Note that the Lua Plugin provides a bridge between the Custom Visualisation Viewer (not shown) and the Lua Interpreter.

5.4 Performance Debugging with the Custom Visualisation Viewer

Since all visualisations have in common that they are able to be displayed in real-time, i.e., synchronised with relevant events in an ongoing search, they are also often useful for performance debugging. Performance debugging concerns the effectiveness of the encoding of a given problem. For example, in the meet-pass problem, one of the constraints says that a train may not enter a sector if there is already a train there, even if that train is about to move on. This can be read: “you cannot have a track sector occupied by trains at successive times, except in the degenerate case where it’s one train that does not move,” and formulated in Zinc as:

```
constraint
forall(t in 1..ntrains, x in 2..nsteps) (
  pos[x,t] == pos[x-1,t] \ /
  forall(u in 1..ntrains) (pos[x-1,u] != pos[x,t]));
```

While logically correct, this does not cause all the propagation of constraints to happen that one would expect. In particular, at the start of the search, one would expect the solver to have worked out that train 1 (i.e., the left top-most/red train) has to be in track sector 1 at time 2. However, as can be seen in the visualisation, the propagation effectively places trains 1 and 2 on sectors 1 and 2 at time 1, but somehow fails to remove sector 2 from the domain of $\text{pos}[2, 1]$. While it may be possible to deduce this situation merely from looking at the assignments of variables in the debugger’s data view, it is *far* easier to spot in the custom visualisation (see Fig. 6).

Noting this, we rephrased the above requirement as “No two different trains can occupy the same sector at successive times,”:

```
constraint
forall(t,u in Trains where t != u) (
  forall(x in 2..nsteps) (pos[x-1,u] != pos[x,t]));
```

The expected propagation then happens. In our experiments, we have found that custom visualisations like this are not only useful to display results, but also to detect performance bottle-necks that stem from weak propagations like these.



Fig. 6. The missing train: pictures of meet-pass at the start of a search. Propagation should deduce the position of the train at the second timestep (top picture), but in the original formulation (bottom picture) it failed to do so.

6 The Solving Layer

The Solving Layer is comprised of three components: the solver platform itself, a *solver monitor* which collects detailed information about the solving process, and a *debugging interface* which is used to control the stopping and starting of the solving process. Fig. 7 provides an overview of our architecture. The solver monitor and the debugging interface run in separate processes; communication between them is achieved by way of an XML-based messaging system where each message is validated against an external XML Schema². The same schema also defines the set of supported Zinc data types which the monitor recognises and tracks during search.

Conceptually, the communication between solvers and the IDE is purely event-driven. Debugger events are defined separately from the solver code. A number of pre-defined API calls are inserted at the relevant points in the solver to trigger the events if a debugging run has subscribed to them; that is, enabled their sending by the solver when they occur. Events that are subscribed to may simply update the debugger's search model or may also suspend the search until the user decides to continue. At such breakpoints, the front-end debugging tools query the internal model of the search for the information they need by following the XML-based communication protocol between the IDE and

² <http://www.w3.org/XML/Schema>

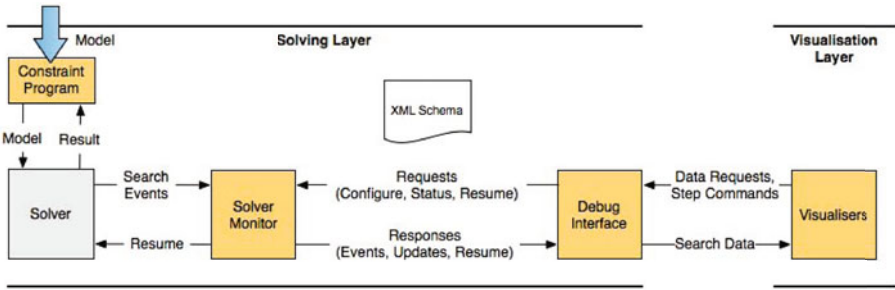


Fig. 7. A closer view of the solving layer. The debugger may be invoked once a model is passed to the constraint program. Note however that the debugger never communicates with the solver directly; rather all communication is via the solver monitor. In addition to controlling the starting and stopping of the solving process the debugger is also responsible for forwarding incremental updates about progress of the solver to the various visualisers.

the solver monitor. The solver adhering to the solver monitor’s API, however, does not have to implement this protocol, which is already handled by the solver monitor.

Our two-process system is highly flexible; by removing any direct dependencies between the solver and interface we are able to transparently substitute one solver for another. A further advantage arising from such a separation is that the solving process can be executed on a completely different (and possibly faster) machine than the one which is running G12IDE. It is entirely possible therefore to commoditize the solving program as an online “service” to which users submit constraint models and from which they receive solutions that can be later visualised in domain-specific terms.

6.1 The Solver Monitor

Invoking the solving process with a debugging flag enables the solver monitor that is attached to the solver. The monitor adheres to a “certainty principle”: it observes the solving process without affecting its course in any way. Before solving commences, a two-way communication channel is opened between the monitor and the debugging interface. We use separate sockets for the requests to and responses from the monitor. This avoids latency problems if a large number of requests need to be sent at one time.

There are three kinds of requests made to the solver monitor: *configuration requests* that modify the state of the monitor by setting and resetting print or break conditions, *status requests* that ask for information about the state of the solver (when stopped) and *resume requests* that cause the solver either to step to the next event or continue until reaching a break condition. Configuration requests may be specific to a particular variable or constraint but can equally apply to an entire class of events. For example, it is possible to pause the search when the solver splits on a particular variable or on any arbitrary variable. Meanwhile a status request can be used to find out the domain of a particular variable or to query the solver about some aspect of its search strategy.

Responses from the monitor include: *solver events*, *status updates* and messages indicating when the solver has stopped. Status updates are generated in direct response

to status requests. Solver events meanwhile are generated whenever the monitor detects that the solving process has reached a point corresponding to a print or break condition. For example, it is possible to generate a solver event message in response to changes to variables and constraints, progression through the search (such as the beginning or end of a propagation phase, or upon reaching a choicepoint or fixpoint) and on the creation of new variables and constraints.

One significant challenge in designing the solver monitor was to resolve the following dilemma: in order to provide maximum flexibility to the visualisation components we need to be able to report detailed information about fine grained search events; at the same time we need to limit the communication between the monitor and the debugging interface as the amount of generated data could easily grow to gigabytes. Our solution is for the solver monitor to maintain an internal table of print and break conditions. Each entry in the table determines what data to send when the solver generates a corresponding search event. Initially the table is empty, optimising the monitor for the “do nothing” case. However, when the solver is paused, configuration requests from the debugging interface are able to modify the table and add (or remove) print or break conditions as required. This way, if a custom visualisation only looks at part of the model it need not incur any overhead associated with processing unrelated events.

6.2 The Debugging Interface

The debugging interface (Fig. 8) provides a variety of mechanisms for controlling the search process and inspecting the model instance under evaluation. A number of “stepping” commands, each operating at varying levels of granularity, are available to progress the search to any given point. Analogues of familiar debugging commands such as “step into”, “step over” and “step return” are available to direct the solver to the next event, the (beginning and end of the) next propagation phase and the next solution respectively.

The familiar Eclipse debugging views, which traditionally show information about the call stack of the current program and the variables on its heap, have been adapted to display information about the solving process. Variables and constraints are shown in an expanding tree, with attributes associated with each such as variable domain or constraint status (whether it is awake, asleep or killed). After each stepping command attributes that have changed are highlighted, similarly to the highlighting of changed data structures in other Eclipse debuggers.

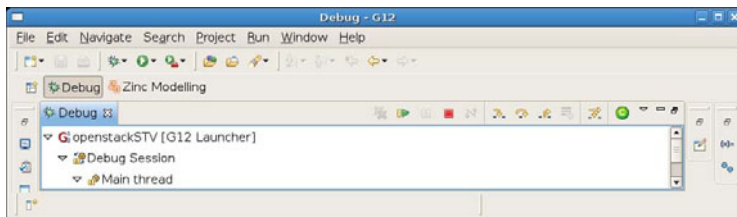


Fig. 8. The debugging controls

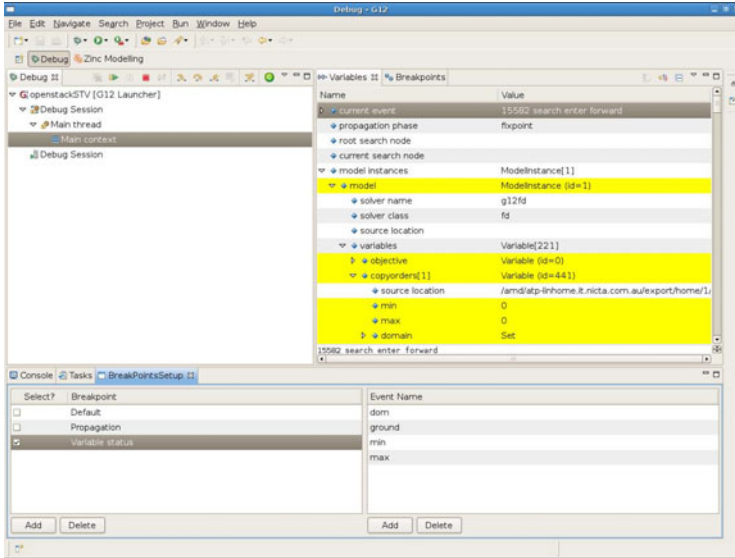


Fig. 9. User-defined breakpoints view. The **C** button advances the solver to the next event specified by the user.

In addition to the pre-defined debugging commands, the IDE allows for custom debugging commands definition, to include only search events that are of interest for the user. For instance, the user might be interested only in changes made to the bounds of decision variables. In such a case, they can create a new *breakpoint* that contains only the “min” and “max” events. The bottom left part of Fig. 9 shows an example of such a breakpoint, called “Variables changed“, whereas the bottom right part shows the search events that the breakpoint is made of. The tick mark next to the breakpoint name shows which breakpoint is currently enabled. Pressing the **C** button sends a request asking for a break every time one of the events belonging to the selected breakpoint occurs.

7 Related Work

There already exist a number of IDEs to support constraint programming systems, some of which provide facilities for visualising constraints in various ways. ECLIPSE CLP [3] for example, allows the modeller to draw a range of charts and graphs of the solver output, whereas ILOG OPL Studio [2] offers a search tree view similar to the one offered by the G12IDE. Choco [1], on the other hand, is more geared towards Java developers who will write their own visualisation tools from scratch, and integrate the Choco Java library as a solver.

There has been work on the use of abstract views of constraints and search to support debugging of constraint programs. The most systematic account is probably that of the DiSCiPl project [4] about a decade ago, which produced a sophisticated search tree viewer and a number of other tools for use with CHIP. OzExplorer [14] was developed at about the same time.

Declarative debugging [11] for Prolog and other logic programming and functional programming languages is related to our approach and could be applied to CP search. It has been studied over many years and many tools were developed. However, it is focused on correctness debugging, which in our experience is rarely the main concern in Zinc modelling. Moreover, the existing tools generally operate at a lower level than those we have developed for G12: we regard debugging a G12 constraint program as a very different activity from debugging the Zinc model.

Drawing tools producing scriptable SVG images are of course well known, and some of them are much more highly developed than the rather simple one we provide. The use of custom views as a way of presenting solutions is also quite standard [9]. However, coupling such a generic tool, including the Lua script editor, to the rest of the debugging package is new with G12IDE.

8 Summary and Conclusions

We presented G12IDE, a front-end for the G12 platform aimed at helping users create and work with constraint models in a manner independent from any underlying solver. Besides offering users an intuitive interface for the G12 platform, it provides advanced features such as debugging a running search and the (custom) visualisation of this process.

Debugging search is in several ways fundamentally different from classical debugging of code. We first had to create a mental model of this process, i.e., define what the relevant concepts and notions are to make this process useful and technically feasible, before attempting an implementation as part of our IDE. Our model is based on a solver monitor which notifies the IDE when relevant events, defined by the user, have occurred in the search to pass control back to the IDE. When this happens, users have the freedom to choose between pre-defined visualisations also known from other, similar IDEs, but also to run their own in this step-wise manner. While supporting this novel model of debugging, we saw it as important to stick as closely as possible to the interface of a classical debugger in order to leverage the application of this new idea. It is also worth noting that there is no opposition between this form of debugging and classical offline profiling: both are useful and one can use both to get a better understanding of how a problem gets solved. Tighter integration of profiling and debugging tools is, in fact, subject to future work and therefore beyond the scope of the present paper.

Custom visualisations, as we have demonstrated using the example of the meet-pass trains, are a useful tool not only to demonstrate high-level and rather coarse properties of a constraint problem, such as the size of its domains, but also to highlight internals of the actual solving process, such as the order of constraint propagation which takes place. As such, visualisations are a useful tool for debugging.

To the best of our knowledge, these core features together with the extensibility of the G12IDE are unmatched in similar front-ends to constraint programming systems. G12IDE is available from the central MiniZinc homepage currently located at the address <http://www.g12.cs.mu.oz.au/minizinc/>.

Acknowledgements. We thank the NICTA G12 team for their valued help and assistance in the development of this work.

References

1. Choco Constraint Solving Toolkit., <http://www.emn.fr/z-info/choco-solver/>
2. ILOG OPL Studio, <http://www.ilog.com/products/oplstudio/>
3. Apt, K., Wallace, M.: Constraint logic programming using ECLiPSe. Cambridge University Press, Cambridge (2007)
4. Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.): DiSCiPl 1999. LNCS, vol. 1870. Springer, Heidelberg (2000)
5. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Bruynooghe, M. (ed.) LOPSTR 2004. LNCS, vol. 3018, pp. 214–232. Springer, Heidelberg (2004)
6. Gent, I.P., Miguel, I., Rendl, A.: Tailoring solver-independent constraint models: A case study with ESSENCE and MINION. In: Proc. 7th Symposium on Abstraction, Reformulation and Approximation (SARA), pp. 184–199 (2007)
7. Harel, D., Koren, Y.: A fast multi-scale method for drawing large graphs. *J. Graph Algorithms Appl.* 6(3), 179–202 (2002)
8. Ierusalimsky, R., de Figueiredo, L.H., Filho, W.C.: Lua—an extensible extension language. *Softw. Pract. Exper.* 26(6), 635–652 (1996)
9. Jones, C.V.: Visualization and Optimization. Kluwer, Boston (1996)
10. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Estimating search tree size. In: Proc. of the Twenty-First National Conference on Artificial Intelligence (AAAI). AAAI Press, Menlo Park (2006)
11. Lee, N.: A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3) (April 1997)
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
13. Quigley, A.J., Eades, P.: Fade: Graph drawing, clustering, and visual abstraction. In: Graph Drawing, pp. 197–210 (2000)
14. Schulte, C.: Oz explorer: A visual constraint programming tool. In: Proc. of the 14th International Conference on Logic Programming (ICLP), pp. 286–300. MIT Press, Cambridge (1997)
15. Wallace, M., The G12 team: G12 - Towards the Separation of Problem Modelling and Problem Solving. In: van Hoes, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 8–10. Springer, Heidelberg (2009)