An Explanation-Based Constraint Debugger

Aaron Rich, Giora Alexandron, and Reuven Naveh

Cadence Design Systems
{ari,giora,rnaveh}@cadence.com

Abstract. Constraints have become a central feature of advanced simulation-based verification. Effective debugging of constraints is an ongoing challenge. In this paper we present GenDebugger - an innovative GUI-based tool for debugging constraints. GenDebugger shows how a set of constraints is solved in a step by step flow, allowing the user to easily pick out steps that are key to the buggy behavior. It is unique in that it is designed for dealing with various types of constraint-related problems, not only constraint-conflicts or performance issues. GenDebugger utilizes principles from the explanation-based debug paradigm [1], in that it displays information justifying each step of the solution. A unique contribution is that it enables the user to easily combine separate explanations of individual steps into a comprehensive full explanation of the constraint-solver's behavior. GenDebugger was lately released for full production, after an extensive process of evaluation with early access users. Our own experience and our users feedback indicate that GenDebugger provides highly valuable help in resolving problems found in constraints.

Keywords: Constrained-Random Verification, Constraint Debugging.

1 Introduction

Constrained Random generation of stimuli is a technique widely used in advanced simulation-based verification. To verify a Device Under Test (DUT) test scenarios are generated randomly but nevertheless must comply with constraints set by the verification engineer. This technique is supported by various verification-oriented languages such as *e* and System Verilog. Coding errors in constraints are revealed in several forms, such as unexpected values assigned to variables, unexpected distribution of values across a set of simulations, or failure to find a solution due to conflicting constraints. Bad runtime and memory consumption of constraint solving activity is also a common problem.

Debugging of constraints poses a considerable challenge (see [3], for example) because constraints reflect complex relationships between variables, and might depend on complicated conditions. What is more, regular debugging tools such as source-line debuggers, with which most software engineers are familiar, are ill-suited for debugging constraints. The reason is that source-line debuggers are normally sequential, showing and controlling the line-by-line imperative flow of user code. Constraints, in contrast, are declarative entities, analyzed and solved by a constraint solver built within the verification software. Because of this the processes governing constraint-solving are not visible to the user and this renders basic concepts of the common debugger paradigm, such as breakpoints and single-stepping, useless.

One approach to constraint debugging is to remove or add constraints until the generated stimuli is satisfactory. This is not necessarily a viable solution, given that it doesn't explain the problem with the original, preferred set of constraints. A second approach is to have the solver print the constraints directly involved in a constraint conflict, but this isn't appropriate for other types of problems. Even for constraint conflicts, it provides very limited help when a complex set of constraints is involved. The human mind can only deal with a limited amount of information at once. Another common debugging method is to trace the constraint solver's activity [2] to a file or the screen, but again - the multitude of static information a user must deal with makes this a very limited solution. A relatively new and promising direction is explanation-based debugging, where a debugger displays information justifying decisions of the constraint solver [1].

2 GenDebugger - Explanation-Based Debugging

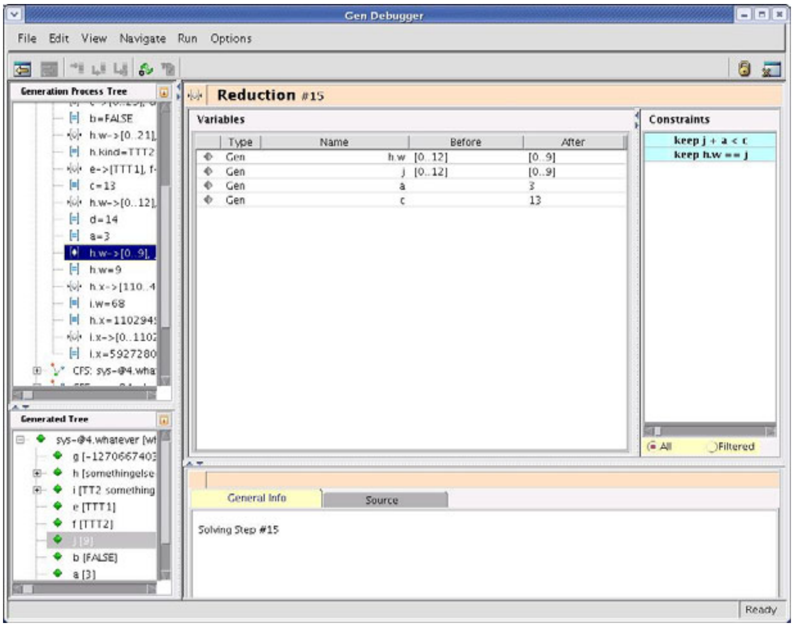
In this paper we describe GenDebugger, the generation debugger of IntelliGen, the new generation engine of the testbench automation tool Specman. GenDebugger depicts the constraint-solving process as a sequence of steps, each reducing the domain of one or more variables, until each variable has a single value that complies with the constraints. Several types of steps are shown, including domain reductions based on constraints, value assignments, backtracking, input sampling, and more.

GenDebugger shows each step in a detailed, interactive view. The variables and constraints involved in the step are displayed, so that it can be clearly seen which variables were reduced and which constraints were involved. Because solving steps typically reduce the domains of variables, the post-step domains are displayed together with the pre-step domains, so that the nature of the change in a variable's domain is evident.

GenDebugger is interactive in that any variable or constraint can be selected and queried. A dedicated panel shows various pieces of information such as the type, path, or source of a selected variable, or the source and declaring class of a selected constraint. A specially important piece of information for a variable is its list of steps (see section 2.2).

2.1 GenDebugger and Explanations

GenDebugger explains solving-steps. That is, it displays information justifying the step, usually a subset of constraints and the domains of related variables. For example, a reduction of the domains of the unsigned integers x and y to



```
b=FALSE
h.w->[0..21],
h.kind=TTT2
e->[TTT1], f-
c=13
h.w->[0..12],
d=14
a=3
h.w->[0..9],
h.w=9
h.x->[110..4
i.w=68
h.x=1102949
i.x->[0..1102
i.x=5927280
CFS: sys-@4.wha
```

```
sys-@4.whatever [wh
g [-1270667403
h [somethingelse
i [TT2 something
e [TTT1]
f [TTT2]
j [9]
b [FALSE]
a [3]
```

Reduction #15

Variables

Type	Name	Before	After
Gen	h.w	[0..12]	[0..9]
Gen	j	[0..12]	[0..9]
Gen	a		3
Gen	c		13

Constraints

```
keep j + a < c
keep h.w == j
```

Solving Step #15

Fig. 1. Explaining a Reduction

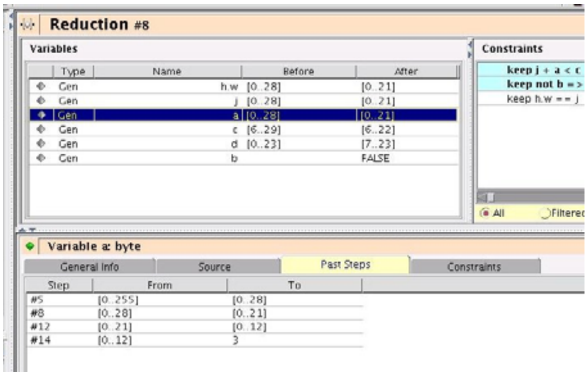
[1..9] and [0..8] respectively, is justified by the constraints *x<10* and *y<x*. After assigning *5* to *y*, a later step further reducing *x*'s domain to [6..9] is explained by the constraint *y<x* together with the value assigned to *y*.

Naturally, a solving step is easier to explain when it involves only a few bits of information. Because of this, when showing a solution to a large set of constraints, GenDebugger breaks the solution into several smaller steps, each much more explainable than the entire process as a whole. See figure 1 to see how GenDebugger explains a single reduction.

2.2 Full Explanations

It is common that some of the elements of a step's explanation are themselves solving steps, performed earlier in the solving process. For example, a reduction on *z* to [0..4], explained by the constraint *z<x-1* coupled with *x*'s domain: [6..9], raises the question: why was the domain of *x* [6..9] in the first place? The explanation is not complete without understanding the earlier reduction on *x* setting it's domain to [6..9].

GenDebugger is designed to enable the user to easily get a *full explanation* for a step. It does this by providing easy access to other, potentially relevant, steps. When any variable involved in the current step is selected, all steps affecting this variable are listed (see figure 2). A simple double-click displays the explanation for this second step. In this way the user combines explanations of several related steps into a full and comprehensive explanation. To continue our example, when viewing the reduction on z, selecting x lists all steps performed on x, including the reduction to [6..9]. Double-clicking on this earlier reduction, causes GenDebugger to display its explanation. The combination of these two explanations together form the full explanation for the reduction of z to [0..4].



Reduction #8

Variables

	Type	Name	Before	After
	Gen	h.w	[0..28]	[0..21]
	Gen	j	[0..28]	[0..21]
	Gen	a	[0..28]	[0..21]
	Gen	c	[6..29]	[6..22]
	Gen	d	[0..23]	[7..23]
	Gen	b		FALSE

Constraints

keep j + a < c
keep not b =>
keep h.w == j

[✓ All] Filtered

Variable a: byte

Step	From	To
#5	[0..255]	[0..28]
#8	[0..28]	[0..21]
#12	[0..21]	[0..12]
#14	[0..12]	3

Fig. 2. All Steps on Variable a

3 Debugging Various Constraint-Related Problems

3.1 Understanding a Constraint Conflict

A constraint conflict is depicted by GenDebugger as a failed attempt to reduce variable domains due to contradicting constraints. When faced with a constraint conflict, the user sets a breakpoint telling GenDebugger to open when the conflict occurs (e.g., *break on gen error*). Then, when opened, GenDebugger automatically displays the failing step's explanation. The explanation can be explored further to form a full explanation, as described above in 2.2.

3.2 Understanding Value Assignments

Sometimes a variable is assigned a value that seems to the user unreasonable. In this case the user can set a second type of breakpoint telling GenDebugger to open when this variable's value is generated (e.g. *break on gen field packet.data*). It then opens to show how the variable is generated. Selecting the assignment-step from the list of steps on the variable, the user gets the required explanation. This can be continued to other, related steps until a full explanation of the unexpected assignment is formed.

Another way to have GenDebugger display the assignment to a variable, after it occurred, is via a simple command (e.g. *show gen x*). If information on the generation of the variable is accessible at that time, GenDebugger opens and automatically displays the explanation for the assignment.

3.3 Understanding Distribution

Distribution issues arise when, across many tests, the entire range of values generated for a variable, or the probability of certain values, is unexpected. To debug this type of issue, the user can utilize the methods described in the previous section 3.2. The user explores what determined the value assigned to the variable in a specific test. This exploration should reveal the constraints blocking, or restricting the probability of, some values from being assigned.

3.4 Understanding Performance

When the system seems stuck on a constraint-solving process, the user can break the test and open GenDebugger. GenDebugger opens on the solving step currently performed by the solver, and the user can also easily see all the steps preceding it. This is likely to reveal what is requiring so much solving activity, and why.

A common reason for performance issues is numerous backtracking. When this occurs, GenDebugger shows many backtrack steps in its list of solving steps. The explanation of a backtrack step is a bit different. It is made of the entire sequence of steps that were cancelled by the backtrack, and that terminated in a failure. GenDebugger shows the cancelled track of steps, allowing the user to understand why it ended at a dead end.

4 GenDebugger Experience

GenDebugger is the culmination of more than 10 years of experience in debugging constraint-solving issues. It was designed based on user feedback, with various types of verification needs. GenDebugger was lately released for full production after an extensive process of evaluation and validation with early access customers. The overall reaction to GenDebugger is no less than enthusiastic. According to our users, there is no comparison between GenDebugger and the earlier constraint-debugging solutions. We feel this is a promising avenue for significantly reducing the effort of creating constraint-based verification environments.

References

1. Ghoniem, M., Jussien, N., Fekete, J.D.: VISEXP: Visualizing Constraint Solver Dynamics Using Explanations. In: Seventeenth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2004, Miami Beach, FL. AAAI press, Menlo Park (May 2004)
2. Meier, M.: Debugging Constraint Programs. In: Saraswat, V., Hentenryck, P.V. (eds.) Principles and Practice of Constraint Programming. LNCS, vol. 976, pp. 204–221. MIT, Cambridge (1995)
3. Simonis, H., Aggoun, A.: Search-Tree Visualisation. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 191–208. Springer, Heidelberg (2000)