Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification

Reuven Naveh and Amit Metodi

Cadence Design Systems, Israel
{rnaveh,ametodi}@cadence.com

Abstract. Constraint programming (CP) figures prominently in the process of functional hardware verification. The verification process is based on generating random tests according to given set of constraints. In this paper. we introduce IntelliGen, a propagation based solver, and the random generator of Cadence's Specman verification tool. IntelliGen is designed to handle several problems beyond the mere need to find a feasible solution, including: generating random tests with a 'good' distribution over the solution space; maintaining test reproducibility through different run modes and minor code changes; and debug of the solving process by verification engineers. We discuss the advantages of CP solvers over other solving technologies (such as BDD, SAT or SMT), and how IntelliGen overcomes the disadvantages of CP.

1 Introduction

Constraint programming (CP) is a major component in functional verification. Functional verification, tests a hardware device using a simulation of the design-under-test (DUT) behavior. Verification is performed by generating diverse random stimuli to produce interesting test scenarios, and collecting coverage to ensure that all aspects of the DUT have been tested. Constraint solvers are used to produce valid test cases that will satisfy the DUT's restrictions [6,5].

Typically, CP seeks a feasible solution as fast as possible. Functional verification changes this focus. Constraint problems in functional verification are not necessarily hard to solve, and often many solutions exist. Instead of seeking a single suitable solution, it is crucial to solve the problem many times with different solutions. Moreover, the random solutions should be well distributed over the (often enormous) solution space, and fulfill user-defined coverage requirements [7].

Several solving technologies are available to handle the random-constrained problems of verification environments. These include CP solvers, SAT solvers, and BDD solvers, as well as hybrid methods. BDD solvers are especially efficient in finding a solution with uniform distribution over the solution space [12]. CP solvers utilize the generation of a random value from a variable's domain and the randomization of variable selection ordering, to control the random selection [3]. SAT solvers are usually used to find one solution, but there are techniques that are used to generate random solutions such as XORSample [2].

C. Schulte (Ed.): CP 2013, LNCS 8124, pp. 823–831, 2013.

An inherent part of the verification process is debugging. When generation leads to a conflict, or results do not satisfy the DUT expectations, the functional verification engineer must debug the constraint solving. This requires the solving process to be random stable: the generation must be reproducible. Typically, the functional verification engineer is not an expert in constraints and does not have actual knowledge in the underlying algorithms used by the solver. Hence, the debug tools should be designed and the solver chosen, with this in mind.

This paper focuses on two of the special needs in the domain of functional verification: distribution of the solutions, and random stability, and shows the advantages of using a CP solver instead of other solving techniques to address these (and other) needs. It also describes the disadvantages of a CP solver, and focuses on how IntelliGen, a Cadence-supplied constraint solver and random generator used by the Specman tool, overcomes those disadvantages.

2 IntelliGen

In the domain of functional verification, test scenarios are generated from user environments using constraint solvers. Cadence is an Electronic Design Automation (EDA) company which, among other things, supplies tools for functional verification. One of these tools, Specman, provides a framework and environment for the verification language *e* [10]. IntelliGen, a Cadence-supplied constraint solver and random generator used by the Specman tool, is a powerful solver that can handle a variety of constraints, including arithmetical, bit operations, and several global constraints (sum, count, all-different, etc.).

The user environments generated by IntelliGen may be quite large and include millions of lines of code, and tens of thousands of constraints and variables. The constraints may be on scalar fields, but can also be used to bind pointers or to decide which subtypes will be generated.

Because of the inter-dependencies inherent in the structure of the environment, it cannot be solved as a single-constraint problem. Therefore, IntelliGen analyzes the entire environment, and separates it into isolated solving problems, denoted as CFSs (Connected Field Sets). Dependencies between CFSs are computed to determine the correct order of solving, and for each CFS, a proper solving device is chosen, depending on the nature of the problem.

Test scenarios may contain thousands of CFSs, and may require that some of the CFSs be solved many times while others may be solved only once. The solving device therefore should ideally be reusable and have a relatively low build cost. In meeting this need, CP solvers have an important advantage over other solving techniques such as BDD or SAT.

IntelliGen also includes GenDebugger, a GUI debugging tool used to debug constraints [1,8]. GenDebugger is intended for use by the verification engineer, typically a software or hardware engineer with knowledge of procedural code debugging. GenDebugger allows debugging of constraint conflicts, unexpected generated results, or performance and distribution problems. It groups the debug information for each CFS, and represents the process of constraint solving as a set of relatively small steps. The steps present information about variables domains (before and after the step), and the relevant constraints that participated in the step. GenDebugger also includes breakpoints that let the user stop at specific points during the generation process.

Using a CP solver is particularly helpful for debug because it facilitates translation of the solver's main operations (such as "value-selection," "propagation," and "backtrack") into a sequence of elementary solving steps that can be described in terms known to the user ("randomization," "reduction," and "value cancellation" respectively). Because each solving step is directly related to specific constraints, the user can be informed which constraints led to which change.

Till now, we briefly discussed IntelliGen and some of the reasons (reuse, relative low-cost build, and easy operation translation for the users) for choosing a CP solver. In the next two sections, we describe two additional aspects (Distribution and Random Stability) of finding a good random solution, the advantage of a CP solver for handling these aspects, and how IntelliGen addresses them.

3 Distribution

The fundamental practice in functional verification is validating the DUT through many randomly generated tests, with a goal of achieving full coverage [7]. To achieve this goal, the tests should be varied, and the randomly generated values should be well distributed over the solution space.

A natural question is "What constitutes 'good distribution'?", and an intuitive answer might be uniform distribution over the solution space, an approach embedded in the SystemVerilog IEEE standard [9]. The concept of uniform distribution is clarified in [6]. There, the authors call for the tests to be distributed as uniformly as possible among all possible tests that conform to the constraint model, the intent being to reach a significantly different solution each time the same constraint model is solved. Thus, uniform distribution of all possible tests is not required, because two different solutions might still create very similar tests. Much more important than uniformly sampling the solution space is sampling all 'interesting' values of the different variables.

An important consideration is that generation problems in verification are usually asymmetric: a typical problem includes 'flags', which control the topology and conditions of the problem, and also as including generated scalar variables, whose domains may vary from very small to very large (e.g., all values of 32-bit unsigned integer). In these cases, the uniform distribution approach is hazardous because some values of the topology variables or small domain variables may not be generated because of their low probability. However, when the solution space is symmetric and comprised of variables with similar initial domains, a uniform or close to uniform distribution is the correct solution.

Consider Example 1, which demonstrates a case where an 'interesting' value for one of the variables will never be generated in a realistic test scenario.

Example 1. Assume two variables x and y with the domains $\{ 1..4 \}$ and $\{ 1..2^{32} \}$ respectively, and a single constraint $(x = 1 \rightarrow y = 2)$.

The solution space of this problem is $\{\,(1,2)\,\} \cup \{\,(a,b)\,|\, a \in \{\,2..4\,\}, b \in \{\,1..2^{32}\,\}\,\}$. When uniformly picking a random pair from this space, the probability for generating the value one for x is $1/(3 * 2^{32} + 1)$. In other words, an 'interesting' value for one of the variables will likely never be generated in a realistic test scenario. This will lead to unfilled areas in the user's coverage.

While unfeasible to generate all values of all variables, at the very least all four values for variable x should be generated. This will lead to non-uniform distribution of the values of y (which are 2 whenever x is 1). The user's expectations can thus be summarized as follows: For variables with a relatively small domain, all possible values are generated, while the distribution should be as uniform as possible for the rest of the model, as long as permitted by the constraint model.

Verification languages, such as *e* or SystemVerilog, include directives to control the distribution of specific fields, but even when no such directives are given, it is preferred that the generated values will vary and be distributed well. In this paper we do not discuss user directives, but only how generation results are naturally distributed depending on the solver technology.

In IntelliGen, a propagation-based solver, the distribution is the result of variable and value ordering. IntelliGen's variable ordering primarily picks a variable with the smallest domain (according to the first-fail principle of Haralick & Elliott [4]). If several variables have the same (or close enough) domain size, the solver chooses the variable randomly. Once a variable is chosen, the generated value is randomized uniformly from its domain. Though intended for finding a solution more quickly, variable ordering based on the first-fail approach also often leads to the desired distribution.

Finding a satisfying assignment for Example 1 when using IntelliGen's variable and value ordering is done in the following solving steps: (1) Choose the variable x (due to its smaller domain), and randomize a value from its domain (which is $\{\,1..4\,\}$). (2) Propagate (reducing y to $\{\,2\,\}$ if x was randomized to 1). (3) If y was not reduced to $\{\,2\,\}$, choose the variable y, and randomize a value from its domain.

This flow (which we term 'randomization-propagation' flow) does not lead to a uniform distribution on the solution space, but it does ensure full coverage on the values of x (which are generated with equal proportions). As for the values of y, they will be varied enough once x was determined. This leads to the conclusion that CP heuristics, intended mainly for finding a solution efficiently, also contribute to achieving good, albeit not uniform, distribution.

As an enhancement to this behavior consider Example 2 which was extracted from a real life verification environment.

Example 2. Assume four variables *delay*, *kind*, *size*, and *lock* with the domains $\{\,0..2^{32}\,\}$, $\{\,SINGLE, INCR, WRAP\,\}$, $\{\,BYTE, HALFWORD, WORD\,\}$, and $\{\,true, false\,\}$ respectively, and a set of constraints:

$$(kind = WRAP \rightarrow lock = true) \qquad (size = WORD \rightarrow kind = WRAP)$$
$$(kind \in \{\,SINGLE, INCR\,\} \rightarrow lock = false) \qquad (lock = true \rightarrow delay = 0)$$

Experimental results of using IntelliGen to generate 10000 items with these constraints show 1437 generations of $(size = WORD)$, and 5141 generations of

($lock = true$). On the other hand, in 10000 generation using uniform-distribution BDD solver, no item at all has ($size = WORD$) or ($lock = true$). Therefore, IntelliGen's behavior meets the user's expectations, which would be that all possibilities of *kind*, *size*, and *lock* will be generated.

A different principle regarding generation distribution can be defined as the unwillingness to generate values without a real cause for choosing them. If we look at the constraint ($x = 1 \rightarrow y = 2$) from Example 1, the value 1 for x should not be biased just because it is mentioned in a constraint. An unneeded repetition of values may lead to tests that are not varied enough, and to an uneven and biased distribution.

Example 3. Assume two variables x and y with the domains $\left\{1..2^{32}\right\}$ for both variables, and a single constraint $(x = 1 \rightarrow y \neq 1)$.

Applying 'randomization-propagation' flow to Example 3 will create the following steps: (1) Choose a variable (either x or y) and randomize its value. (2) If the chosen variable was randomized to one, omit one from the other variable's domain. (3) Choose the remaining variable and randomize its assigned value from its current domain. Observe that the impact of the constraint is minimal. The probabilities of x and y to be 1 are close to $1/2^{32}$, and therefore these values will be generated similarly to any other values in the variables' domains, and with very low probability.

A generation of biased values may occur when using an SMT solver over a CP theory. Most SMT solvers are designed to determine a single satisfying solution, but attempts have been made to adapt them generate random results [11]. The constraint $(x = 1 \rightarrow y \neq 1)$ is equivalent to the SMT formula $(\neg(x = 1) \vee (y \neq 1)))$ which is translated to $(\neg b1 \vee b2)$ where the atoms $(x = 1)$ and $(y \neq 1)$ are replaced by Boolean variables $b1$ and $b2$ respectively. The values for the Boolean variables are assigned by the SAT solver. Any of the three valid possibilities for $(b1, b2)$, which are (*false*, *false*), (*false*, *true*) and (*true*, *true*), are acceptable. However, the value *true* for $b1$ or the value *false* for $b2$ enforces that x will be 1 or y will be 1, correspondingly. This differs from the CP behavior, which gave no additional weight to these values, and they were generated with very low probability, like the rest of the values in the variables' domains. This drawback of SMT solvers can also be seen in Example 4, which is a variation of Example 2.

Example 4. Suppose we add the constraint ($kind = SINGLE$) to Example 2.

Now there is no constraint enforcing *delay* to become zero. Indeed, in 10000 generations using IntelliGen, no zeros were generated for this variable. However, the SMT solver, when the SAT layer has a literal for ($delay = 0$), might occasionally assign *true* to this literal, leading to biased distribution. Thus, we see that the CP solver, and in particular the 'randomization-propagation' flow, maintains the principle of not generating biased values. The SMT behavior violates this principle and is much more sensitive to addition and removal of constraints, even when their influence on the environment is minor.

The examples thus far have demonstrated the positive aspects of CP solvers in achieving a good distribution when using first-fail variable ordering and uniform distribution value ordering. But as mentioned before, the 'randomization-propagation' flow does not always achieve uniform distribution. In cases of variables with symmetrical domains, such distribution might be the desired one, as can be seen in Example 5.

Example 5. Assume two variables, x and y, with the domains $\{ 0..9 \}$ for both variables, and a single constraint $(x + y < 10)$.

Let us compare the probabilities of the solutions $(9, 0)$ and $(0, 0)$ for (x, y). The solution $(9, 0)$ has 5.5% probability: 10% if x is chosen first, and 1% if y is chosen first. The solution $(0, 0)$ has 1% probability. Since the solution space contains 45 solutions, a uniform randomization should choose each solution with 2.22% probability. In Example 5, there is no reason to prefer one solution over the other, and each combination of values can represent an interesting test-case. Assuming the solution space should be fully covered, this means that many data items will need to be generated until the coverage criterion is satisfied. The problem becomes much more serious when it involves many variables and not just two, as Example 6 illustrates.

Example 6. Assume a list m of variables with the domain $\{ 0..2^{32} \}$, and a single constraint $(\sum m_i = 10000)$.

Initial propagation over this constraint cannot narrow the initial domain of the variables beyond $\{ 0..10000 \}$. Therefore, the first chosen variables will be generated relatively freely, enforcing the remaining variables to sum to a rapidly diminishing value. In a typical solution very few variables are randomized to values between zero and ten thousand, several more are small positive numbers, and the rest are zeros. This is definitely not the user's expectation of a solution.

Starting from Specman version 11.1, IntelliGen incorporated a more sophisticated randomization scheme regarding the sum global constraint, while still remaining loyal to the 'randomization-propagation' flow. While the main details cannot be shared, the main principle is that the variable's value is not randomized uniformly from the variable's domain, but rather it is chosen according to the expected solution for the whole list. The generated lists do satisfy the criterion of being uniformly distributed.

Using the latest Specman and IntelliGen, the experimental results for Example 6, with 10000 generations of a list of 100 variables, show: 6302 generations have a list which contains a zero, 2677 generations have a list which contains two zeros, and only six generations have a list which contains more than five zeros.

Contrast this with the naive 'randomization-propagation' flow prior to Specman 11.1: In 10000 generations, all results contain more than 70 zeros, and 4845 generations contain more than 90 zeros.

To summarize, the two main principles of distribution are to generate all the interesting test scenarios and to avoid biased generations of values. CP solvers using the 'randomization-propagation' flow satisfy these principles, unlike other solving techniques. However, constraint models which are not handled well by the CP solver should receive special attention by the solver.

4 Random Stability

Once a test involves picking a random solution, the requirement to maintain reproducibility of the generation results becomes crucial. This is especially true in the verification flow, which may look as follows: (1) A nightly regression catches a failure in a specific test case. (2) A verification engineer reproduces the test case and chooses its owner. (3) The owner runs the test case again, using debug or interactive tool. (4) A fix is implemented and validated by rerunning the test in the regression.

Each of these steps involves a different run mode: regression tests run in batch mode, while the user will often prefer to use GUI tools. In addition, when debugging the test, devices such as breakpoints or logging may be used. These different run modes might involve different settings (for example, the memory setting of the run might be influenced). Yet, it is highly important that the same generated results will be produced in each run. Any difference, even the smallest, may drastically change the flow of the test, leading to irreproducibility.

Maintaining reproducibility is a difficult task when using non-deterministic algorithms; naive parallel search or propagation algorithms do not ensure it. However, even in a totally deterministic generation, it is not a trivial matter to maintain reproducibility, because the machine environment and settings may influence random stability. The following scenario demonstrates a case in which a different behavior of garbage collection changes random stability:

A test generates a data item ten times. During the nightly run this test failed on DUT error in the tenth generation. The solving was done using a learning solver (e.g. SAT solver) and no garbage collections were performed. When debugging the test, the different settings increased its memory signature, so that garbage collection was issued before the tenth generation. The garbage collection may have removed some of the learned data, or even the entire solver.

In this scenario, where learning was used, part of the learned data was removed in the debug run, but in the nightly run all solvers and the corresponding learned data was kept. Therefore this scenario will produce different results for the tenth generation in each run mode of the test. The lesson is that even a deterministic solver is vulnerable to random stability problems, and that learning solvers are especially prone to such problems.

While absolute reproducibility is essential, verification tests have an even more pressing requirement: Code modifications that are irrelevant to the generation of a certain variable must not affect its generated values when using the same seed. However, using one seed will obviously make the generation very sensitive to code changes. To support the random stability requirement, many more seeds are required.

IntelliGen expands the concept of 'seed' and uses a multi-seed scheme. After breaking the constraint model into isolated solving problem CFSs, a unique seed is attached to each CFS and to each CFS variable. The CFS seed is used, for example, to randomize variable ordering when solving the CFS. This ensures that independent CFSs will keep random stability regarding each other. Each CFS variable has its own seed which is used to randomize values from its domain. This is done to improve random stability for the specific variable. Sometimes random stability may be kept even when code changes are done within the same variable's CFS, although it cannot be guaranteed, as Example 7 illustrates.

Example 7. Assume three variables x, y and z with the domains $\{1..2^{32}\}$ for all variables, and a constraint $(x \leq y)$.

Adding the constraint $(y \leq z)$ will affect the generation of x, although its domain is unchanged. Because the randomization results are dependent on variable ordering, if z is chosen first, the generated value of x is different. However, using propagation based solving ('randomization-propagation' flow used in IntelliGen as described in Section 2) is sometimes helpful in maintaining random stability even when code changes are made to closely related variables. The reason is that generation of values is done using the variable domains with their dedicated seeds. This is illustrated by Example 8.

Example 8. Assume three variables b, x and y, with the domains $\{0..1\}$, $\{1..2^{32}\}$, and $\{1..2^{32}\}$ respectively, and the constraints: $(b = 1 \rightarrow x < 100)$, and $(b = get\_flag\_value())$ where $get\_flag\_value()$ is a runtime method which may reflect the current status of the DUT.

Adding the constraint $(b = 1 \rightarrow y < 50)$ will not affect the generation of the variable x, although it is related to the same CFS with the variables y and b. This is because the domain of x remains the same after propagation, with or without the additional constraint. Note that a different randomization approach, for example one which generates a vector of solutions, will generate different values for x with or without the additional constraint. Translating the problem to a BDD is an example. The reason is that the additional constraint reduces the solution space, as it reduces the domain of y.

Maintaining random stability, in all its aspects, is one of the most important requirements IntelliGen is expected to support. Choosing a CP solver and using the multi-seed scheme enables IntelliGen to support this requirement.

5 Summary

This paper discusses the benefits of a CP framework to address the special requirements of functional verification. The requirements of the domain are beyond the need for finding a single feasible solution, and rely heavily on the randomness of the solutions. These benefits were one of the main motivations for the technology picked for Cadence's IntelliGen constraint solver. The considerations include: the ability of the solver to efficiently solve the constraint problems, the random quality and stability of the solutions, and the convenient way the results can be presented and reproduced for debug. CP solvers prove to be highly effective for these problems, and show advantages over other solving techniques.

References

1. Alexandron, G., Lagoon, V., Naveh, R., Rich, A.: Gendebugger: An explanation-based constraint debugger. 2010 TRICS, 35 (2010)
2. Carla, A.S., Gomes, P., Selman, B.: Near-uniform sampling of combinatorial spaces using xor constraints. In: Proceedings of NIPS 2006 (2006)
3. Dechter, R., Kask, K., Bin, E., Emek, R.: Generating random solutions for constraint satisfaction problems. In: AAAI/IAAI, pp. 15–21 (2002)
4. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14(3), 263–313 (1980)
5. Lagoon, V.: The challenges of constraint-based test generation. In: Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, pp. 1–2. ACM (2011)
6. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcu, E.S., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine 28(3), 13 (2007)
7. Piziali, A.: Functional verification coverage measurement and analysis. Springer (2004)
8. Rich, A., Alexandron, G., Naveh, R.: An explanation-based constraint debugger. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 52–56. Springer, Heidelberg (2011)
9. Society, I.C.: IEEE Standard for System Verilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std. IEEE (2005)
10. I. C. Society. IEEE Standard for the Functional Verification Language 'e'. IEEE Std. IEEE (2006)
11. Wille, R., Große, D., Haedicke, F., Drechsler, R.: Smt-based stimuli generation in the systemc verification library. In: Borrione, D. (ed.) Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's. LNEE, vol. 63, pp. 227–244. Springer, Heidelberg (2010)
12. Yuan, J., Albin, K.: Simplifying boolean constraint solving for random simulation-vector generation. In: Proceedings of ICCAD, pp. 412–420 (2004)