CRAVE 2.0: The Next Generation Constrained Random Stimuli Generator for SystemC

Hoang M. Le, University of Bremen, Bremen, Germany (*hle@informatik.uni-bremen.de*)

Rolf Drechsler, University of Bremen and DFKI GmbH, Bremen, Germany (*Rolf.Drechsler@dfki.de*)

***Abstract*— CRAVE is an open-source constrained random verification environment for SystemC. Since the first release, CRAVE is constantly under active development. The paper introduces the next major release CRAVE 2.0 containing many novel features, which increase the practical usefulness of CRAVE significantly. The focus is on three main improvements: soft constraints, distribution constraints and constraint partitioning.**

Keywords—SystemC; constrained random verification; soft constraints; distribution constraints; Satisfiability Modulo Theories;

I. INTRODUCTION

Constrained Random Verification (CRV) [1] has become the prevalent method for functional verification due to its advantages over traditional directed testing approaches. Creating a huge number of test vectors manually is not required anymore. Instead, these values will be automatically generated from the legal input space defined by a set of constraints. Furthermore, CRV can reveal bugs using corner-case stimuli that might have not been thought of.

Meanwhile, SystemC [2] has gained acceptance as a unified language for both design and verification of electronic systems at multiple levels of abstraction. This is partly owing to the *Accellera Systems Initiative* (ASI) open-source reference simulator[1] that has made SystemC widely available. In the same vein, ASI also provides the *SystemC Verification Library*[2] (SCV) as an open-source CRV solution for SystemC. However, despite the recent update of SCV 2.0, its capacities are still lacking in comparison with tools for dedicated *Hardware Verification Languages* (HVLs) such as SystemVerilog [3] or e [4]. The most severe problem with SCV is that its constraint solver is still based on *Binary Decision Diagrams* (BDDs) [5], which is unable to handle large sets of complex constraints involving many variables.

At the end of 2011, we have released the first version of CRAVE[3], which is to the best of our knowledge the only other open-source CRV implementation for SystemC. CRAVE provides many improvements over SCV such as a better API for constraint specification and management, automatic constraint debugging, etc. (see [6] for more details). Most importantly, the constraint solver of CRAVE is based on modern *Satisfiability Modulo Theories* (SMT) solvers [7], which is well-known for their much better scalability in comparison to BDDs.

Since the first release, CRAVE is constantly under active development. In this work, we present the next major release CRAVE 2.0 introducing many technical enhancements, with some of them being highlighted below:

1. Support for soft constraints: Soft constraints are of great use for verification engineers to specify default behaviors (e.g. default values) which can be modified later (e.g. in the subsequent class specializations). When soft constraints cause a conflict, only the ones responsible for the conflict should be ignored. Users of the HVL e and recently SystemVerilog[4] can use this feature to the full extent, while SCV and CRAVE 1.0 just drop all soft constraints in case of conflict. CRAVE 2.0 removes this limitation.

[*]This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

[1] SystemC 2.3 (includes TLM), http://www.accellera.org/downloads/standards/systemc

[2] SystemC Verification Library 2.0, http://www.accellera.org/downloads/standards/systemc

[3] Available at http://www.systemc-verification.org/crave

[4] After the major update of IEEE 1800-2012 [3]

2. Support for distribution constraints: Both SCV and CRAVE 1.0 do not allow distribution constraints to be mixed together with other constraints. Instead, the values of variables will be generated according to specified distributions in the first step. Then, the constraint solver is invoked to solve the remaining constraints using these generated values. This approach can lead to randomization failure, which is undesirable. On the other hand, dealing with distribution constraints is very challenging for SMT solvers, since they are geared to generate only one solution. CRAVE 2.0 implements a pragmatic approach by treating distribution constraints as a special class of soft constraints, i.e. CRAVE 2.0 guarantees to generate valid stimuli if they exist, while trying to be as close to the specified distributions as possible.

3. Constraint Partitioning: A real-world verification environment can contain tens of thousands of constraints. Trying to solve them at once would create a major performance bottleneck. Fortunately, these constraints can often be partitioned into many smaller constraint sets, which can be solved independently. CRAVE 2.0 makes use of this observation to boost its performance considerably.

These newly implemented features, being absent in both SCV and CRAVE 1.0, increase the practical usefulness of CRAVE significantly.

II. Preliminaries on CRAVE

This section briefly introduces the relevant basics of CRAVE to make the paper self-contained. For simplicity, many features are omitted (which can be found in [6]).

A. *Constraint Specification*

In CRAVE, every *random object* (e.g. data packet, whose content is to be randomized) is an instance of a user-defined class which extends the base class *rand_obj*. This user-defined class can have as members other random object classes or *random variables* of primitive types (e.g. C++ *int* or SystemC *sc_uint*). A random variable of type *T* is defined as an instance of the template class *randv<T>*. Constraints on random variables can be specified in constructors of the class using the construct *constraint* or *soft_constraint*, respectively. The following example shows a random packet with two member variables to be randomized. The destination address must always be in the valid range [0, 0xFFFF000], while the packet size is constrained to be in [10, 999]. The difference between *constraint* and *soft_constraint* will be explained later in Section III.

```
class packet : public rand_obj {
    randv<unsigned int> size;
    randv<unsigned int> dest_addr;
    packet() : rand_obj() {
        constraint(dest_addr() <= 0xFFFF0000);
        soft_constraint(size() >= 10);
        soft_constraint(size() < 1000);
    }
};
```

B. *Constraint Solving*

In order to generate values respecting the specified constraints, CRAVE converts the constraints into an SMT formula and employs an SMT solver. Different SMT solvers are supported via a unified interface provided by metaSMT [8]. While an SMT solver generally supports advanced features such as uninterpreted functions or quantifiers, CRAVE only needs the support for bit-vectors (often referred to as the QF_BV theory), which is very stable, well-optimized and well-tested. The main advantage of using SMT solvers instead of BDDs as in SCV is that SMT solvers scale much better (see the experimental evaluation of [6] for an example). However, with SMT solvers it is much more difficult to control the distribution of the generated values, since SMT solvers are optimized to generate just one solution while BDDs implicitly represent all solutions. Different heuristics can be employed [9] if a uniform distribution is desired. However, in many cases, a uniform distribution is not the best option. This will be discussed further in Section IV.

III. Soft Constraints

The construct *constraint* is used to specify *hard constraints* which must always be satisfied if possible. On the contrary, soft constraints, being specified using *soft_constraint*, can be ignored if they conflict with hard constraints or other soft constraints. The main use of soft constraints is to specify default behaviors (e.g. default values) which can be modified later (e.g. in the subsequent class specializations). For further details on the use of soft constraints in functional verification, we refer to [10] where a methodology is described.

Now consider the packet example, we want to verify how the DUT handles short packets whose size is between 5 and 9. An e or SystemVerilog user would define a new class *short_packet* that inherits from *packet* and adds two more soft constraints as shown in the following code. These two new soft constraints cause a conflict with the soft constraint *size() >= 10* and the desired behavior is to drop this constraint so that short packets can be generated. In SCV or CRAVE 1.0, however, all soft constraints will be dropped leaving the variable *size* unconstrained. As a result, very long packets can be generated instead of the desired short packets.

```
class short_packet : public packet {
    short_packet() : packet() {
        soft_constraint(size() >= 5);
        soft_constraint(size() < 10);
    }
};
```

Essential for the use of soft constraints is a well-defined dropping scheme for them. The basic idea is to assign a unique priority to each soft constraint and when the constraints contradict each other, dropping a soft constraint with lower priority is preferred. Consequently, the priorities should be assigned in a way that soft constraints in subsequent classes are higher prioritized. Recall that in CRAVE, the constraints are created at runtime (during the construction of a constrained object), it is very natural to assign priorities based on the order of constraint creation, since a constructor of a specialized C++ class will be called after the corresponding constructor of the base class. More precisely, we keep a counter on how many soft constraints have been created, and each time a new soft constraint is added, its priority is assigned to be the current value of the counter before it is incremented. For the example, the priorities will be assigned as follows. First, the constructor *short_packet()* is called to create a short packet. Then, this constructor calls the constructor *packet()*, which in turn, adds the first two soft constraints from the packet class with priority 1 and 2, respectively. After that, the body of *short_packet()* is executed resulting in the creation of two new soft constraints with priority 3 and 4, respectively. Because these four constraints contradict themselves, the constraint with lowest priority (i.e. *size() >= 10* with priority 1) is to be dropped. The following pseudo code shows how this behavior is realized in the general case.

```
Inst = an empty SMT instance;
foreach (hard_constraint) { add hard_constraint to Inst; }
if (!satisfiable(Inst)) // the hard constraints contradict themselves
    report and show contradictions to user;
foreach (soft_constraint) { add soft_constraint to Inst; }
if (!satisfiable(Inst)) { // all hard and soft constraints cannot be satisfied together
    foreach (soft_constraint) { remove soft_constraint from Inst; }
    foreach (soft_constraint) in descending order of priority {
            add soft_constraint to Inst;
            If (!satisfiable(Inst)) { // this soft constraint causes a conflict
                    remove soft_constraint from Inst;
                    report to user that soft_constraint has been dropped;
            }
    }
}
```

Note that the repeatedly solving of the SMT instance (i.e. the call *satisifiable(Inst)* in the pseudo code) together with the addition and removal of constraints can be efficiently implemented using the incremental solving capability available in modern SMT solvers. Also, the priority-based dropping scheme for soft constraints of CRAVE 2.0 aligns very well with the semantics of soft constraints in SystemVerilog [3] easing a potential switch of language.

IV. Distribution Constraints

As mentioned in Section II, it is difficult to control the distribution of generated values using SMT solvers, especially when a uniform distribution is desired. However, a uniform distribution is not always the best distribution as demonstrated by the following example.

```
class my_rand_obj: public rand_obj {
    randv<unsigned int> a, b, c;
    my_rand_obj() : rand_obj() {
        constraint(if_then(a() < 10, c() == 0));
        constraint(if_then(b() < 10, c() == 1));
        constraint(a() <= 1000000000);
        constraint(b() <= 1000000000);
        constraint(c() <= 1000000000);
    }
};
```

An object of the class *my_rand_obj* contains three non-negative integers *a*, *b* and *c* in range [0, 10^9]. The first two constraints specify two implications: if *a* (or *b*) is smaller than 10, then *c* must be 0 (or 1, respectively). These constraints make the solution space extremely asymmetric. For a constraint solver, which generates uniformly distributed solutions, the possibility to generate an object where either $a < 10$ or b < 10 is extremely small. But from the coverage-driven verification point of view, it is necessary to also consider these values.

There are several solutions to tackle this issue such as changing the variable ordering (e.g. using the construct *solve before* in SystemVerilog) or allowing user-defined biases. In CRAVE 2.0, we choose to support user-defined biases using distribution constraints. For example, we can specify that for *a*, 30% of the generated values should be < 10 and for *b*, 50% of the generated values should be < 10. The new construct *dist* to specify these distributions in CRAVE 2.0 is shown in the following class extending *my_rand_obj*.

```
class my_rand_obj_ext: public my_rand_obj {
    my_rand_obj_ext() : my_rand_obj() {
        constraint(dist(a(),
            distribution<unsigned int>::create
                (weighted_range<unsigned int>(0, 9, 30)) // 30%
                (weighted_range<unsigned int>(10, 1000000000, 70)) // 70%
        ));
        constraint(dist(b(),
            distribution<unsigned int>::create
                (weighted_range<unsigned int>(0, 9, 50)) // 50%
                (weighted_range<unsigned int>(10, 1000000000, 50)) // 50%
        ));
    }
};
```

While it is also possible to specify distributions with SCV and CRAVE 1.0 (although with slightly different constructs), the way these distribution constraints being solved differs significantly. SCV and CRAVE 1.0 solve these separately from other constraints as follows. First, SCV and CRAVE 1.0 generate values for *a* and *b* according to the specified distributions. Now assume that the generated values for *a* and *b* are both in [0, 9]. In the next step, these values are fixed for *a* and *b* when all other constraints are solved together causing a conflict because c cannot be both 0 and 1. As a result, a randomization failure is reported back to the user. This is an undesirable behavior, because the constraint set is solvable.

CRAVE 2.0 implements a pragmatic solution to handle such cases. It tries to solve both distribution constraints and the rest together. In the first step, the values are still generated according to the specified distributions. However, these will not be fixed in the solving step but rather be treated in a similar manner to soft constraints. More precisely, assume there are N variables, whose values have been already generated, if fixing these N values causes a conflict, in the next try, one value will be dropped and (N – 1) values will be fixed, and so on. This is repeated until the constraints can be successfully solved. The main idea behind this procedure is that as few generated values as possible should be dropped so that the actual distribution will be as close as possible to the specified.

For the example, of 10000 objects generated by CRAVE 2.0, we count 2220 occurrences of *a* and 4225 occurrences of *b* from the range [0, 9]. The resulting percentages (22,2% and 42,5%) are quite close to the specified 30% and 50%. The detailed distributions are shown in the following table. As can be seen, within the range [0, 9], the values of both *a* and *b* are evenly distributed.

	0	1	2	3	4	5	6	7	8	9	Total	%
a	230	200	230	234	216	214	228	220	219	229	2220	22,2
b	437	420	446	451	408	406	421	413	410	413	4225	42,3

V. Constraint Partitioning

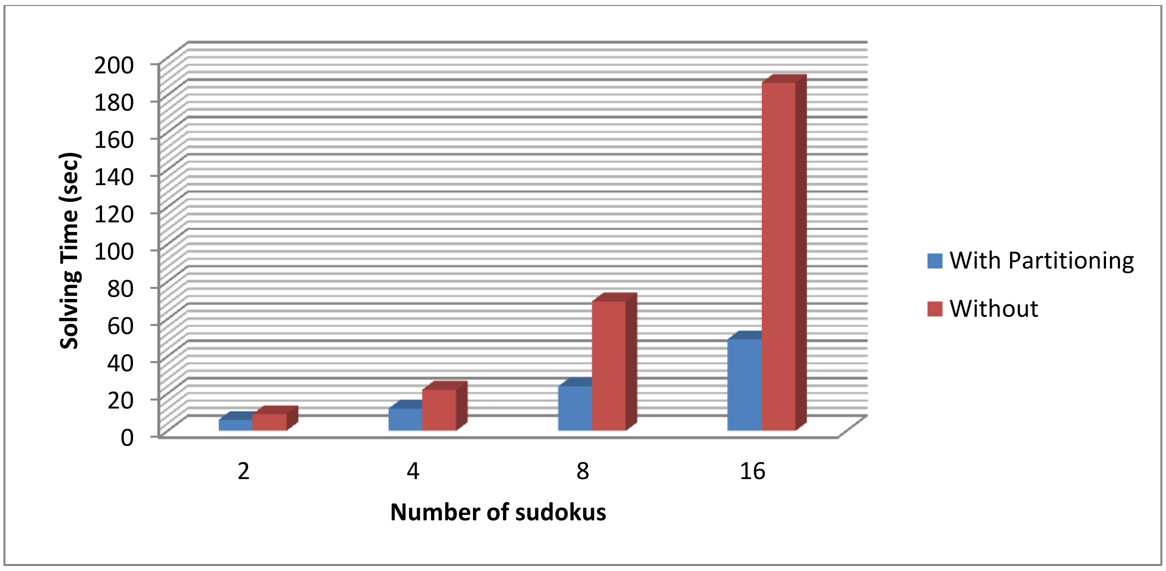
We continue with our packet example by extending it to be a multicast packet. A fixed-size array of addresses of other destinations is added and constraints are also applied to each element of the array to ensure that these addresses are also in the valid space. The constraint set for a multicast packet therefore contains one constraint for the packet size, one constraint for the first destination, and one constraint each for the newly added destinations.

```
class multicast_packet : public packet {
    randv<unsigned int> other_dest_addr[16];
    multicast_packet() : packet() {
        for (int i = 0; i < 16; i++)
            constraint(other_dest_addr[i]() <= 0xFFFF0000);
    }
};
```

It is obvious that the constraints are actually independent, i.e. each variable *size*, *dest_addr*, *other_dest_addr[i]* can be solved separately because the value of each variable does not affect the constraints for the other variables. SCV does not take that into account and tries to build a single BDD for all these constraints and cannot finish within a reasonable amount of time. While this example can be solved instantly by CRAVE 1.0, for more complex constraint sets, significant improvements on performance can be observed for CRAVE 2.0.

Note that it is possible for a user to perform the partitioning manually. But for real-world verification environments which can contain tens of thousands of constraints, it is very time-consuming and error-prone. Furthermore, it also defeats the purpose of having a declarative constraint language. CRAVE 2.0 automatically divides constraints into independent constraint partitions based on the concept of *support sets*. For a single constraint, its support set is the set of all variables contained in the constraint. Consequently, the support set for a constraint partition is the union of the support sets of all individual constraints. Obviously, if the support sets of two constraint partitions are disjoint, each partition can be solved separately by the constraint solver and the results can then be combined to create a complete solution.

CRAVE 2.0 implements a simple algorithm for constraint partitioning. It starts with a partition containing a single constraint and tries to maximize the partition by incrementally adding constraints, whose support sets intersect with the current support set of the partition. If the partition cannot be enlarged anymore, it can be solved by the constraint solver without affecting other constraints, thus a new partition is created with a constraint which does not belong to any existing partition, and the whole process is repeated. A better implementation using more advanced data structures for disjoint sets (e.g. union-find data structure) is left for future work.



Now we demonstrate the performance improvement by constraint partitioning for solving sudokus, which exhibit much more complex constraint sets than our multicast packet. In each experiment, we solve a constraint set containing multiple independent sudokus with and without constraint partitioning. As can be seen from the above diagram, applying constraint partitioning results in clearly better performance and scalability.

VI. Conclusion

The paper presents a new version of the SystemC open-source constrained random verification environment CRAVE. This version, called CRAVE 2.0, is significantly more useful in practice due to the three new highlighted features: support for soft constraints, support for distribution constraints and constraint partitioning. For each of these features, example usage and necessary implementation details are provided to help a potential user in both adopting and extending CRAVE.

Acknowledgment

The authors would like to thank Tino Flenker, Finn Haedicke and Daniel Große for contributing to the base architecture of CRAVE 2.0.

References

[1] J. Yuan, C. Pixley and A. Aziz, Constraint-based verification, 1st ed. New York, NY: Springer, 2006.

[2] 'IEEE Standard for Standard SystemC Language Reference Manual', IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp. 1-638, 2012.

[3] 'IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language', IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009), pp. 1-1315, 2013.

[4] 'IEEE Standard for the Functional Verification Language e', IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008), pp. 1-495, 2011.

[5] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, Vol. C-35, No. 8, August, 1986, pp. 677-691.

[6] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC,". In International Symposium on System-on-Chip, pp. 1-7, 2012.

[7] L. De Moura and N. Bjørner, 'Satisfiability Modulo Theories: Introduction and Applications', *Commun. ACM*, vol 54, iss 9, pp. 69--77, 2011.

[8] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler, "metaSMT: Focus on Your Application not on Solver Integration,". In Proceedings of DIFTS@FMCAD, 2011.

[9] R. Wille, D. Grosse, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC Verification library,". In Forum on Specification & Design Languages (FDL), pp. 1-6, 2009.

[10] M. Strickland, H. J. Zhang, J. Chen, D. Goswami, and A. Wakefield, "Soft Constraints in SystemVerilog: Semantics and Challenges,". In Design and Verification Conference (DVCON), 2012.