CRAVE: An Advanced Constrained RAndom Verification Environment for SystemC

Finn Haedicke[1] Hoang M. Le[1] Daniel Große[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{finn, hle, grosse, drechsle}@informatik.uni-bremen.de

Abstract—**A huge effort is necessary to design and verify complex systems like System-on-Chip. Abstraction-based methodologies have been developed resulting in Electronic System Level (ESL) design. A prominent language for ESL design is SystemC offering different levels of abstraction, interoperability and the creation of very fast models for early software development. For the verification of SystemC models, Constrained Random Verification (CRV) plays a major role. CRV allows to automatically generate simulation scenarios under the control of a set of constraints. Thereby, the generated stimuli are much more likely to hit corner cases. However, the existing SystemC Verification library (SCV), which provides CRV for SystemC models, has several deficiencies limiting the advantages of CRV. In this paper we present CRAVE, an advanced constrained random verification environment for SystemC. New dynamic features, enhanced usability and efficient constraint-solving reduce the user effort and thus improve the verification productivity.**

I. Introduction

Creating a new *System-on-Chip* (SoC) involves many tasks. Once the specification is agreed on, the modeling phase starts to derive a potential solution. To manage the complexity of today's SoCs high-level languages are used for the first models. For this *Electronic System Level* (ESL) design phase [1] a widely accepted approach is SystemC [2], [3], [4], [5]. Integrated hardware and software models can be developed, exchanged and refined based on the SystemC IEEE standard [6]. In particular, *Transaction Level Modeling* [7], [8] allows to create high performance virtual platforms for early software development and architectural analysis. In addition, the high-level models serve as reference to verify the behavior of the more detailed descriptions built in the following stages.

In general, facing today's verification challenges a verification environment needs to be constructed. From a high-level perspective three major components are necessary: stimuli, assertions and coverage. Assertions are used to check the functional correctness and therefore monitor design variables [9]. The task of functional coverage is to measure which design functionality has been exercised during simulation [10]. Both are not in the focus of this work. Here, we target the problem of stimuli generation. But instead of deterministic values as defined in traditional directed testbenches we make use of *Constrained Random Verification* (CRV) [11], [12]. CRV applies input stimuli to the design that are solutions of constraints. These solutions are determined by a constraint-solver. CRV offers two key benefits: First, CRV enables to find unexpected assertion violations since scenarios are simulated which the verification engineer might have not thought of. Second, the stimulus generation process is automated and hence a huge set of scenarios can be executed leading to higher coverage. Hence, for large and complex systems the confidence in the correct functionality significantly increases.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

For SystemC CRV is available through the *SystemC Verification* (SCV) library [13], [14], [15]. However, the SCV library has several deficiencies:

1) Poor dynamic constraint support, hence no control of constraint effects at run-time
2) No constraint specification for dynamic data-structures
3) Low usability when specifying constraints for composed data structures
4) Poor information in case of over-constraining
5) Limits in complexity of constraints, since constraint-solving is based on *Binary Decision Diagrams* (BDDs) [16] only

In this paper we present CRAVE, an advanced *C*onstrained *RA*ndom *V*erification *E*nvironment for SystemC.[1] To overcome the limitations of the SCV library CRAVE provides the following features:

- New constraint specification API
  An intuitive and user-friendly *Application Programming Interface* (API) to specify random variables and random objects has been developed.
- Dynamic constraints and data structures
  Constraints can be controlled dynamically at run-time. Moreover, constraints for elements of dynamic data structures like e.g. STL vectors can be specified.
- Improved usability
  Inline constraints can be formulated and changed incrementally at run-time. Furthermore, automatic debugging of unsatisfiable constraints is supported.
- Parallel constraint-solving
  BDD-based and SAT/SMT-based techniques have been integrated for constraint-solving. A portfolio approach is used to enable very fast generation of constraint solutions.

Please note the usage of CRAVE is not limited to pure hardware designs. For example, the constraint solutions can also be used to describe software tests running on a SoC. In the experiments such an example is presented.

For an industrial application of CRAVE we refer to [17].

[1]CRAVE is freely available (w/ source code) under MIT license at www.systemc-verification.org.

The rest of this paper is structured as follows: Related work is discussed in Section II. Section III presents the API of CRAVE. Then, in Section IV the dynamic features of CRAVE are introduced. The usability aspects are described in Section V. Section VI presents the constraint-solving approach of CRAVE and Section VII compares CRAVE to the SCV library in an experimental evaluation. Finally, the paper is concluded in Section VIII.

II. Related Work

As mentioned above, the CRV techniques of the SCV library have several weaknesses which limits their use in practice. Therefore, several improvements for the SCV library have been developed. In [18] bit-vector operators have been added and the uniform distribution among all constraint solutions is ensured in all cases. An approach to determine the exact reasons in case of over-constraining has been presented in [19]. In [20] the BDD-based constraint-solver is replaced by a method which uses a generalization of *Boolean Satisfiability* (SAT).

However, all these approaches compensate only some of the SCV weaknesses. In particular, no constraints on dynamic data structures can be specified, constraints cannot be controlled dynamically during run-time, references to the state of constraints are not available, and no inline constraints are possible restricting the usability. In addition, the integration of different constraint-solvers working in parallel is mandatory to reduce the time for stimuli generation to a minimum.

To standardize verification processes the so-called *Universal Verification Methodology* (UVM) has been developed [21]. Essentially, UVM is a methodology and a class library for building advanced and reusable verification components. The initial implementation has been done for SystemVerilog. Meanwhile UVM also provides a SystemC class implementation. However, it does not include CRV (which is the core verification technique in the UVM methodology).

III. Constraint Specification

In this section we describe the basics of CRAVE. In particular this includes the APIs to create random variables and constrained random objects.

A. Random Variable

From the user's point of view, the most elementary entity is the template class *randv<T>*, which corresponds to a random variable of the C/C++ or SystemC built-in type *T*. All standard applicable operators (arithmetic, comparison, logical, etc.) are overloaded so that an instance *x* of *randv<T>* behaves as if it were a variable of type *T*. A call *x.next()* assigns a random value in the range of *T* to *x*. While the focus of CRAVE is on complex constraints involving many variables, it also supports simple constraints on a single variable. Two member functions *addRange* and *addWeightedRange* can be used to refine the distribution of *x.next()*. Furthermore, *x()* returns a symbolic link to the value of *x* to be used to specify constraints in conjunction with other instances of *randv<T>* as shown in the next section. Table I summarizes exemplarily the API of *randv<int>*.

```
 1  struct packet : public rand_obj {
 2   randv< unsigned int > src_addr;
 3   randv< sc_uint<16> > dest_addr;
 4
 5    packet() : src_addr(this), dest_addr(this) {
 6     constraint(src_addr() <= 0xFFFF);
 7     constraint("diff", src_addr() != dest_addr());
 8     soft_constraint(dest_addr() % 4 == 0);
 9    }
10  };
```

Figure 1: A basic constrained random packet

```
struct packet1 : public packet {
  randv< char > data;
  packet1() : data(this) {
      constraint('a' <= data() && data() <= 'z');
      constraint(dest_addr() % 2 == 1);
  }
};
```

Figure 2: An inherited packet

B. Random object

Complex constrained random objects can also be specified. They must inherit from the class *rand_obj* provided by CRAVE. Such an object can contain several instances of *randv<T>* and *rand_obj*. Constraints for each of these instances as well as constraints between them can be specified in a constructor of the object. For the instance *x* of *rand_obj*, *x.next()* randomizes all belonging instances of *randv<T>* and *rand_obj*, respecting the specified constraints.

We demonstrate the specification of constraints using an example. Figure 1 shows a constrained random packet consisting of two integers to be randomized: a source address as *randv<unsigned int>* and a destination address as *randv<sc_uint<16>>*. The member instances are forced to register themselves to the *rand_obj* as shown in line 5. The source address is constrained to be in the range [0x0, 0xFFFF] (line 6) and the source address and destination address must not be the same (line 7). Both constraints are so-called hard constraints, i.e. they must be satisfied otherwise *next()* should fail. The second constraint is also a named constraint, which enables dynamic management of constraints as described later in Section IV. Line 8 shows a soft constraint stating that the destination address should be a multiple of four. Soft constraints can be ignored by the constraint-solver if they cannot be satisfied in conjunction with the specified hard constraints. As can be seen in all the constraints, the symbolic links to the actual instances of *randv* are used. This packet will be extended step-by-step and serves as a running example for the rest of this paper. Table II shows a summary of the basic API features of *rand_obj*. Note that while constraints should be specified in a constructor for the most use cases, it is possible to add further constraints to an instantiated object using the API.

Table I: The APIs of *randv<int>*

Description	API
Supported operators	+, −, ∗, /, %, ==, ! =, >, <, >=, <=, !, &&, ||, ∼, &, |, <<, >>, ^
Add range to distribution	*addRange(left, right)*
Add weighted range	*addWeightedRange(left, right, weight)*
Generate random value	*next()* returns *true* on success
Symbolic link	*x()* with *x* being a variable of type *randv<int>*

Table II: The basic APIs of *rand_obj*

Description	API
Add hard constraint	*constraint(expression)*
Add named hard constr.	*constraint(name, expression)*
Add soft constraint	*soft_constraint(expression)*
Randomize the object	*next()* returns *true* on success

C. Constraint Inheritance

The inheritance/reuse of constraints in CRAVE is straightforward. The user can add more fields and constraints to an existing random constrained object by using C++ class inheritance. Figure 2 shows an extension of the packet introduced in the last section. Line 2 adds a data field to the packet. The constraint for this data field is declared in line 4. The destination address of the packet is further constrained to be an odd integer on line 5. This new hard constraint contradicts the soft constraint specified on line 8 of Figure 1 and therefore renders the soft constraint useless.

The APIs of CRAVE described in this section are the basics to specify constrained random objects. Similar APIs are also available in the SCV library to a greater or lesser extent. However, CRAVE enables an enhanced usability in comparison to the SCV library as discussed in Section V. In the next section we introduce the distinctive features of CRAVE regarding dynamic constraints and data structures.

IV. Dynamic Constraints and Data Structures

This section introduces three distinctive features of CRAVE which are not supported by the SCV library: constraints on dynamic data structures (currently only vector is supported), dynamic enabling/disabling of constraints, and the concept of references that allows the randomization to interact tightly with the verification environment.

A. Vector Constraints

The SCV library offers no direct support for the constrained randomization of dynamic data structures such as vectors, lists and trees. The user must mimic dynamic data structures by using arrays of fixed-size. This is inconvenient and not memory-efficient. Furthermore, the upper-bound on size of dynamic data structures might not be known at the time of constraint specification. CRAVE offers a template class *rand_vec<T>* for the constrained randomization of vectors. Currently only C/C++ and SystemC built-in data types are supported as the template parameter *T*. The class *rand_vec<T>* also implements the APIs of the STL class *vector* and thus behaves as if it is an STL vector. Similar to *randv<T>*, for an instance *v* of *rand_vec<T>*, *v* refers to the actual vector, while *v()* is the symbolic vector used to specify constraints. For the symbolic vector *v()*, *v().size()* refers to the size, *v()[_i]* to a symbolic vector element, and *v()[_i - c]* to a previous element relative to *v()[_i]* (*_i* is a predefined constant in CRAVE and *c* is a positive constant). The symbolic elements *v()[_i]* and *v()[_i - c]* are used in a *foreach* constraint.

Figure 3 shows an extension of our packet with the data field, now being a constrained random vector. The constraint on the vector is declared in line 5. In the next lines, three *foreach* constraints are specified for the vector. The first two ensure that the first element is an upper case letter and the rest are lower case letters. Both are hard constraints. The third constraint (line 14) is a soft *foreach* constraint: two consecutive elements cannot be *aa*, *ab* or *ba*.

B. Dynamic Constraint Management

During the constrained random verification process, it is very useful that the user can enable/disable specific constraints of a random object. This functionality is not available in the SCV library. The user must mimic the feature by adding an auxiliary variable and constrain this variable in an implication with the constraints to be enabled/disabled. Moreover, this is inconvenient and inefficient. In the CRAVE framework, named constraints can be enabled/disabled directly via the constraint management APIs of *rand_obj*: *enable_constraint(name)* and *disable_constraint(name)*. For example, the constraint on line 7 of Figure 1 can be disabled by calling *disable_constraint("diff")*. Disabled constraints will have no effect in the randomization via *next()* until they are enabled again. Note that the vector constraint *foreach* can also be named and intentionally soft constraints cannot be enabled/disabled.

C. References

In many use cases, the randomization depends on the dynamically changing state of the verification environment. Using the SCV library, to include the state in the constraints the user must use additional variables to save the state and update them manually whenever the state is changed. For this purpose, CRAVE provides references as a convenient shortcut. References in CRAVE basically links a "real" variable with a symbolic variable which can be used during constraint

```
struct packet2 : public packet {
  rand_vec< char > data;

  packet2() : data(this) {
     constraint( data().size() % 4 == 0
        && data().size() < 100 );

     constraint.foreach( data, _i, IF_THEN( _i == 0,
          'A' <= data()[_i] && data()[_i] <= 'Z') );

     constraint.foreach( data, _i, IF_THEN( _i != 0,
          'a' <= data()[_i] && data()[_i] <= 'z') );

     constraint.soft_foreach( data, _i,
        data()[_i] + data()[_i−1] > 'a' + 'b');
  }
};
```

Figure 3: An inherited packet using vector constraints

```
packet2(int &expected_max_size) : data(this) {
  constraint(data().size() % 4 == 0
   && data().size() <=
      reference(expected_max_size));
  ...
}
```

Figure 4: Example of CRAVE reference

specification. Before the constraints are solved, the value of this symbolic variable is fixed to the actual value of the linked variable. Figure 4 gives an example for using references: The size of the constrained random vector *data* of the packet in Figure 3 should not exceed the value of environment variable *expected_max_size*, which is constantly changing. As can be seen, the construct *reference* links *expected_max_size* to a symbolic variable and *data().size()* is constrained to be smaller or equal to this symbolic variable.

The new features introduced in this section demonstrated different use cases where CRAVE offers clear advantages in comparison to the SCV library. The next section discusses important usability enhancements of CRAVE.

V. Usability

The CRAVE framework provides several usability enhancements in comparison to the SCV library. In this section we present inline constraints, incremental constraints and automatic over-constraint analysis.

A. Inline and Incremental Constraints

Constraints in CRAVE can be specified without a formal constraint class. A standalone constrained random generator can be created anywhere and used with arbitrary variables and constraints. In practice, this reduces the effort when coding non-trivial testbench environments. Here is a concrete example to demonstrate this feature:

```
1  randv<int> x,y;
2  Generator gen;
3
4  gen( x() < y() );
5  gen( x() > 100 || y() < −100);
6
7  if (gen.next()) run_test(x,y);
```

This example declares two variables `x`, `y` (line 1) and a constrained random generator (`gen`, line 2). The generator can simply be called to add new constraints: the relation of `x` and `y` (line 4) is specified and that either `x` has to be larger than 100 or `y` is less that -100 (line 5) is constrained. To generate values, the `next` function can be called which returns false if the generator is over-constrained, i.e. the constraints are contradictory and hence no solution exists. When the constraint-solver has generated a stimulus, the *randv<int>* variables can directly be used, e.g. in `run_tests`.

A generator can use all features of CRAVE except for inheritance. However, incremental constraint specification is supported. This feature is very helpful in dynamic testbenches. After the generator has been executed for a certain set of constraints, new constraints can freely be added e.g. to generate more general values first and more specific ones later. We exemplify this in Figure 5.

```
1  randv<int> x,y;
2  Generator gen;
3
4  gen(x() != y());
5  for (int i =0 ; i < 1000; ++i) {
6    gen.next();
7    run_test(x,y);
8  }
9
10 gen( x()∗x() == y() );
11 for (int i =0 ; i < 500; ++i) {
12   gen.next();
13   run_test(x,y) ;
14 }
15
16 gen( y()%2 == 0 );
17 for (int i =0 ; i < 500; ++i) {
18   gen.next();
19   run_test(x,y) ;
20 }
```

Figure 5: Incremental constraint modification

This example first uses a general constraint (lines 4-8). Then, additional constraints are added to focus on specific behavior (lines 10-14 and 16-20). Please note in the final loop at line 20 all three constraints are respected by the constraint-solver.

B. Debugging Constraint Contradictions

For large constraint sets it can easily happen that the overall constraint contains contradiction(s). In this case the problem is over-constrained and hence the constraint-solver is unable to generate valid stimuli. Debugging the contradiction manually is very time-consuming. Therefore CRAVE can automatically identify which named constraints are part of a conflict.

As discussed earlier the soft constraint in Figure 1 (line 8) and the constraint in Figure 2 (line 5) form a conflict. If the former constraint would have been declared as a hard constraint, no constraint solution exists. For such situations CRAVE provides an analyzer, that identifies the conflicting constraints and returns their names. The analysis includes each named constraint and checks them against all unnamed constraints, which are always enabled. In a first step the analyzer determines how many and which constraints need to be disabled to resolve all contradictions. In subsequent steps each of these constraints is expanded to determine the "complete" contradiction. For the example, the first step would determine that one contradiction exists and identify the first constraint. In the next step the second constraint is identified. This is completely done on a formal level, therefore the algorithm is complete and will return all minimal subsets of the constraints that form a conflict. For the described example the run-time of the analysis was negligible.

VI. Parallel Constraint-Solving

Various alternatives to BDD-based constraint-solving have been studied, see e.g. [22]. Approaches based on *Boolean Satisfiability* (SAT) [23], [24] or *Satisfiability Modulo Theories* (SMT) [20] have shown to give very good results for constraints which are hard to solve for BDDs. However, in general it is not possible to know in advance which type of constraint-solver will show the best performance. Therefore, CRAVE uses a portfolio approach. Instead of running a specific constraint-solver, an SMT-based constraint-solver as well as a BDD-based constraint-solver are executed in parallel for the same set of constraints. This section describes the basics for constraint solving and how the portfolio approach works. The next section will compares the run-times of CRAVE and the SCV library.

To integrate the most recent reasoning engines we use *metaSMT* [25] for implementing the constraint-solving in CRAVE. Essentially, *metaSMT* allows engine independent programming by providing a unified interface to different solvers. Hence, no algorithmic changes are necessary when switching to another solver. The overall architecture is depicted in Figure 6. As can be seen CRAVE forms the top layer which implements all the features described in the previous sections. This layer connects to the *metaSMT* front-end layer using the unified input language. In the middle-end layer the transformations for optimization and the basic parallelization features (e.g. threading of different engines) is available. Finally, the backend gives access to a wide range of solvers (see e.g. SWORD [26], Z3 [27], Boolector [28], MiniSAT [29], PicoSAT [30], CUDD [31] and AIGER [32]).

Based on these constraint-solving techniques we have made the following observations for parallelization. In a simple portfolio approach each constraint would be evaluated (at least) twice using different solvers in a multi-threaded environment.

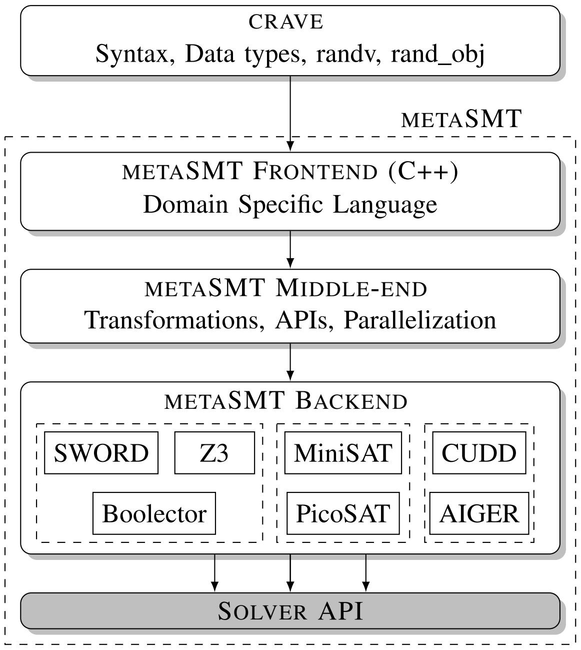

Figure 6: Constraint-Solving Architecture

Then, the result of the fastest solver would be used. However, for CRV a predictable quality of the stimuli is required. Therefore, if a BDD can be built for the overall constraint this is preferred since a uniform distribution among all solutions can be guaranteed[2]. In CRAVE this is reflected by running the SMT-solver only until the BDD is build. From this point on only the BDD is used and execution the SMT solver is stopped.

VII. Experimental Evaluation

In the following examples we demonstrate the advantages of CRAVE for stimuli generation.

A. Arithmetic Constraints

Figure 7 shows a constraint object for a 16 bit ALU (later we scale the size of the ALU). The constraint specifies four operations with their respective input ranges. Table III shows the differences between the classical BDD constraints solver of the SCV and the portfolio approach of CRAVE. The first column gives the name of the library. Two rows are given for both: The first row shows the run-time needed to generate the first solution, and the second row shows the run-time in seconds for the complete execution of the constraint generator, respectively. The following columns provide the data for different bit width of the ALU constraints. As can be seen with increasing bit width of the ALU the SCV fails to solve the constraints. In contrast, in CRAVE the SMT-solver can already generate stimuli before the BDD is ready. Furthermore, note that for ALU16 the 32 bit memory restriction of the SCV

[2]A BDD represents all solutions and hence to select each solution with the same probability is simple. Essentially each path to the 1-terminal needs to be weighted accordingly respecting the reduction rules of reduced ordered BDDs.

```
struct ALU16 : public rand_obj {
  randv< sc_bv<2> > op ;
  randv< sc_uint<16> > a, b ;

  ALU16() : op(this), a(this), b(this) {
   constraint(IF_THEN(op() == 0,
        65535 >= a() + b()) );
   constraint(IF_THEN(op() == 1,
        65535 >= a() − b()
        && b() <= a()) );
   constraint(IF_THEN(op() == 2,
        65535 >= a() ∗ b()) );
   constraint(IF_THEN(op() == 3,
        b() != 0));
  }
};
```

Figure 7: 16 bit ALU constraint

Table III: Comparison of CRAVE and the SCV

		ALU4	ALU12	ALU16	ALU24	ALU32
SCV	first	< 0.01	13.77	MO	TO	TO
	finished	0.09	19.84	MO	TO	TO
CRAVE	first	< 0.01	< 0.01	0.01	0.01	0.01
	finished	0.14	0.30	0.37	0.40	0.49

TO = time out, MO = memory out, run-time in seconds

library was hit. CRAVE can also be build on 64 bit architectures which was however not required for theses experiments.

B. Sudoku Constraints

In the second experiment we formulated the rules of the Sudoku puzzle in CRAVE and the SCV library. Figure 8 gives the respective constraints. In line 4 the 81 random variables of 4 bit size each are declared as the two-dimensional array *res_sdk*. Then, in line 6 the standard C++ two-dimensional array *given_sudoku* is declared which is filled when reading the Sudoku numbers from a file. From line 9 on five types of constraints follow. At first, the numbers of the *given_sudoku* array are assigned to the constraint variables (line 13). Note that we use here the reference feature of CRAVE. Hence, if a solution for the Sudoku-constraint is requested (by calling next), then the current values of *given_sudoku* are used as values for the constraint variables. In other words the standard C++ array *given_sudoku* can be changed anywhere and the actual values will be used in the constraint automatically. Next, the ”obvious” constraints stating a valid solution range for each field (line 15), difference of rows and columns (line 20 and 26) are formulated. Finally, from line 32 on the difference per region is modeled.

For 15 puzzles the constraint-solver had to find a solution (between 16-32 numbers are set in the puzzle; 1 was unsolvable). The run-time (in seconds) and the memory consumption (in MB) were measured with a limit of 2 CPU hours or 4 GB of memory for each Sudoku instance. The evaluation showed largely homogeneous result, hence we only present the aggregated results in Table IV. Although plenty of memory was provided, the SCV library could not solve a single instance[3]. In contrast, CRAVE solved all instances very fast.

```
 1  class sudoku : public rand_obj {
 2  public:
 3    // variable to store solved sudoku
 4    randv< sc_dt::sc_uint<4> > res_sdk[9][9];
 5    // variable to hold given sudoku
 6    int given_sudoku[9][9];
 7
 8    sudoku(rand_obj∗ parent = 0) : rand_obj(parent) {
 9      // constrain given numbers
10      for (int i = 0; i < 9; i++)
11        for (int j = 0; j < 9; j++)
12          constraint( IF_THEN( reference(given_sudoku[i][j]) != 0,
13            res_sdk[i][j]() == reference(given_sudoku[i][j]) ) );
14
15      // only numbers from 1 to 9 are allowed
16      for (int i = 0; i < 9; i++)
17        for (int j = 0; j < 9; j++)
18          constraint((res_sdk[i][j]() >= 1) && (res_sdk[i][j]() <= 9));
19
20      // every number must appear exactly one time in one row
21      for (int i = 0; i < 9; i++)
22        for (int j = 0; j < 9; j++)
23          for (uint k = j + 1; k < 9; k++)
24            constraint( res_sdk[i][j]() != res_sdk[i][k]() );
25
26      // every number must appear exactly one time in one column
27      for (int j = 0; j < 9; j++)
28        for (int i = 0; i < 9; i++)
29          for (int k = i + 1; k < 9; k++)
30            constraint( res_sdk[i][j]() != res_sdk[k][j]() );
31
32      // every number must appear exactly one time in one region
33      for (int i = 0; i < 9; i++)
34        for (int j = 0; j < 9; j++)
35          constraint
36            ( res_sdk[i][j]() != res_sdk[ index(i,1) ][ j ]() )
37            ( res_sdk[i][j]() != res_sdk[ index(i,2) ][ j ]() )
38
39            ( res_sdk[i][j]() != res_sdk[ i ][ index(j,1) ]() )
40            ( res_sdk[i][j]() != res_sdk[ index(i,1) ][ index(j,1) ]() )
41            ( res_sdk[i][j]() != res_sdk[ index(i,2) ][ index(j,1) ]() )
42
43            ( res_sdk[i][j]() != res_sdk[ i ][ index(j,2) ]() )
44            ( res_sdk[i][j]() != res_sdk[ index(i,1) ][ index(j,2) ]() )
45            ( res_sdk[i][j]() != res_sdk[ index(i,2) ][ index(j,2) ]() );
46    }
47  };
48
49  int index(int x, int by ) {
50    return (x + by) % 3 + x − (x % 3);
51  }
```

Figure 8: Sudoku Constraints

Table IV: Comparison of CRAVE and the SCV

Sudoku		min	max	median
SCV	time	2 636.60	4 529.07	3 165.34
	mem	MO	MO	MO
CRAVE	time	0.81	1.83	1.42
	mem	14.00	15.40	14.40

MO = memory out, run-time in seconds

[3]Note these instances are not inherently hard for BDD-based solvers. A dedicated BDD-based Sudoku solving algorithm was able to find the solution for each instance in less than a minute using no more than 216 MB of memory.

```
struct bubble_sort_input : public rand_obj {
  // start address in mem
  randv< unsigned int > start;
  rand_vec< unsigned int > data;

  bubble_sort_input() : start(this), data(this) {
    constraint(0x70 <= start() && start() < 1024);
    constraint(start() % 4 == 0);

    constraint( 0 < data().size()
              && data().size() < 1024);
    constraint(start() + 4 ∗ data().size() <= 1024);

    constraint.foreach(
      data, _i, data()[_i] <= 0x00FFFFFF );
    constraint.foreach(
      data, _i, data()[_i] <= data()[_i−1] + 5);
  }
};
```

Figure 9: Input data constraints for bubble sort

C. Program Input Generation for CPU Testbench

We also apply CRAVE to verify a CISC CPU with 8 registers of 32 bit data width each. The CPU implements a subset of the instructions of the IA-32 architecture including load/store, arithmetic, jump and halt instructions [33]. The CPU is available at three different levels of abstraction: an Instruction Set Architecture (ISA) model in C++, a SystemC TLM model using OSCI TLM-2.0, and a SystemC RTL model implementing a five-stage pipeline [34], [35]. We use CRAVE to generate programs (i.e. instruction sequences) as well as their inputs, which can be used as stimuli for all three models. Then, the simulation-based equivalence checking approach in [35] for models at different levels of abstraction is applied. We describe only one verification scenario: for an instruction sequence implementing the bubble sort algorithm, we randomize its input under the constraints shown in Figure 9. The first four constraints ensure that the array to be sorted fits into the CPU memory and does not collide with the loaded program. The last constraint forces the array to be nearly non-increasing (and thus challenging for bubble sort). Such a concise set of constraints would have not been possible with the SCV library due to the lack of support for dynamic data structures. The average time for CRAVE to generate the first 1000 arrays is approximately 90s (0.09s per array).

VIII. Conclusions

In this paper we have presented the advanced constrained random verification environment CRAVE. After the introduction of the API for constraint specification we have shown the advantages of CRAVE in comparison to the existing SCV library. The advantages include dynamic constraint specification and management, enhanced usability and much faster constraint-solving based on a portfolio approach. All these aspects have been demonstrated by means of examples. In summary, CRAVE improves the verification productivity for SystemC models significantly.

A possible direction for future work is to extend the support for dynamic data structures and to improve the distribution of the SAT/SMT generated stimuli.

References

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[2] Accellera Systems Initiative, "SystemC," 2012, available at http://www.systemc.org.
[3] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.
[4] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., 2005.
[5] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[6] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2005.
[7] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS*, 2003, pp. 19–24.
[8] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.
[9] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.
[10] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.
[11] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Springer, 2006.
[12] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.
[13] *SystemC Verification Standard Specification Version 1.0e*, SystemC Verification Working Group, http://www.systemc.org, 2003.
[14] J. Rose and S. Swan, *SCV Randomization Version 1.0*, 2003.
[15] C. N. Ip and S. Swan, "A tutorial introduction on the new SystemC verification standard," www.systemc.org, White Paper, 2003.
[16] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.
[17] M. F. S. Oliveira, C. Kuznik, W. Mueller, F. Haedicke, H. M. Le, D. Große, R. Drechsler, W. Ecker, and V. Esen, "The system verification methodology for advanced TLM verification," in *CODES+ISSS*, 2012.
[18] D. Große, R. Ebendt, and R. Drechsler, "Improvements for constraint solving in the SystemC verification library," in *ACM Great Lakes Symposium on VLSI*, 2007, pp. 493–496.
[19] D. Große, R. Wille, R. Siegmund, and R. Drechsler, "Contradiction analysis for constraint-based random simulation," in *FDL*, 2008, pp. 130–135.
[20] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *FDL*, 2009, pp. 1–6.
[21] *Accellera Systems Initiative - Universal Verification Methodology 1.1*, http://www.accellera.org, 2011.
[22] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrainted random simulation," in *Int'l Conf. on CAD*, 2007, pp. 258–265.
[23] S. M. Plaza, I. L. Markov, and V. Bertacco, "Random stimulus generation using entropy and XOR constraints," in *DATE*, 2008, pp. 664–669.
[24] H. Kim, H. Jin, K. Ravi, P. Spacek, J. Pierce, B. Kurshan, and F. Somenzi, "Application of formal word-level analysis to constrained random simulation," in *CAV*, 2008.
[25] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler, "metaSMT: Focus on your application not on solver integration," in *DIFTS'11: 1st International workshop on design and implementation of formal tools and systems*, 2011, pp. 22–29.
[26] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "Sword: A SAT like prover using word level information," in *VLSI of System-on-Chip*, 2007, pp. 88–93.
[27] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340.
[28] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *TACAS*, 2009, pp. 174–177.
[29] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.
[30] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
[31] F. Somenzi, *CUDD: CU Decision Diagram Package Release 2.4.1*. University of Colorado at Boulder, 2009.
[32] "Aiger," http://fmv.jku.at/aiger/.
[33] *IA-32 Architecture Software Developer's Manual*, Intel Corporation, 2003.
[34] A. Biere, D. Kroening, G. Weissenbacher, and C. Wintersteiger, *Digitaltechnik - eine praxisnahe Einführung*. Springer, 2008.
[35] D. Große, M. Groß, U. Kühne, and R. Drechsler, "Simulation-based equivalence checking between SystemC models at different levels of abstraction," in *ACM Great Lakes Symposium on VLSI*, 2011, pp. 223–228.