Construction of Concrete Verification Models from C++

Malay Haldar, Gagandeep Singh, Saurabh Prabhakar, Basant Dwivedi, Antara Ghosh

Calypto Design Systems
2933 Bunker Hill Lane, Suite 202
Santa Clara, CA 95054

ABSTRACT

C++ based verification methodologies are now emerging as the preferred method for SOC design. However most of the verification involving the C++ models are simulation based. The challenge of using C++ for sequential equivalence checking comes from two aspects (1) Language constructs such as pointers, polymorphism, virtual methods, dynamic memory allocation, dynamic loop bounds, floating points pose difficulty in creating a model suitable for equivalence checking (2) The memory and runtime required for creating models suitable for equivalence checking from practical C++ designs is huge.

In this paper we describe techniques for constructing verification models from C++ designs containing a very rich set of language constructs. The flow is built keeping in mind that formal methods are inherently capacity constrained but need to be applied to large C++ designs to have practical value.

Categories and Subject Descriptors

B.6.3 [**LOGIC DESIGN**]: Design Aids—*Verification*

General Terms

Design, Verification

Keywords

Formal Verification, C++, Pointers, Dynamic Memory Allocation, Equivalence Checking

1. INTRODUCTION

The necessity to compress the design cycle and develop software concurrently with hardware is driving the methodology where a golden C++ model is created for the SOC. Apart from hardware software co-design, the C++ model acts as an executable specification to the hardware to be designed and helps in exploring various design options for the hardware. Hence it is of immense value to prove that the designed hardware is equivalent to the golden C++ model for all inputs of interest. Since the set of all inputs of interest is very large, a practical approach is to intelligently choose a set of test vectors to accomplish the goal. However, using sequential equivalence checking techniques, one can indeed assert that the golden C++ model and the hardware implementation for it are equivalent for all inputs. In cases where exact equivalence is not of interest, other properties could be proven which hold true for both the golden C++ model and the implemented hardware. The problem lies in handling real life C++ models. The C++ golden models are written with two objectives in mind 1) fast simulation speed and 2) flexibility to explore multiple architectures. Due to these objectives, C++ models typically have heavy usage of pointers, dynamic memory allocation, polymorphism and floating point computations. All of these pose difficulty in applying formal techniques which have been developed for hardware descriptions that lack these language constructs. Further, application of formal techniques to the entire C++ model in one go is not practical because of capacity constraints.

We solve this problem in a stepwise fashion by applying divide and conquer. The first step is the construction of a concrete verification model. The reason we refer to it as a concrete model is that it represents the exact functionality of the input C++ design. This is to differentiate from abstract verification models [1] which are simplified models of the C++ design. The concrete verification model is essentially a finite state machine which represents the given C++ design in a bit-accurate and cycle-accurate manner. Similarly, for the given revised design, which could be another C++ design or an RTL implementation of the C++ design, we synthesize a finite state machine. We then try to locate possible intermediate equivalences between the two state machines through techniques other than formal verification. Finally formal techniques are used in a controlled manner to prove or disprove the equivalences. Note that the two models given for comparison can be either C++ designs or RTL, leading to a variety of applications for the whole verification flow.

In this paper we will only focus on the construction of the finite state machine from C++ designs. We will not be discussing the details of the rest of the verification flow. However, to discuss the experiments and results we will be using the complete equivalence checking setup in some cases.

The rest of the paper is organized as follows : Section 2 surveys related work and describes our contributions in that context. Section 3 describes the details of constructing the verification model from C++. Section 4 describes some experimental results followed by conclusions in Section 5.

2. RELATED WORK AND SUMMARY OF CONTRIBUTIONS

Our techniques are related to work done in three separate fields - formal verification, behavioral synthesis and compiler analysis. Our main contribution is in building a framework that is capable

DAC 2008 June 8–13, 2008, Anaheim, California, USA.

of taking real life C++ programs and applying equivalence check on them. To the best of our knowledge we are not aware of any equivalence checking framework that is capable of handling pointers without any restrictions, dynamic memory allocation, polymorphism, virtual methods, function pointers and floating point arithmetic. Next we discuss some of the proximities to these fields and our extensions and differences with the known techniques.

2.1 Pointer Synthesis

While constructing the concrete verification model, we synthesize a state machine from the given C++ design. Behavioral synthesis or high level synthesis also constructs a state machine from a given C++ design. So there are similarities between our work and behavioral synthesis. However, the objective of behavioral synthesis is to optimize the resulting finite state machine for area, timing and power. Our goal is to optimize the state machine for verification purposes. The difference in optimization criteria leads to different techniques being adopted. Specifically, synthesizing pointers and complex datastructures in C++ has been dealt in [2]. The goal in behavioral synthesis is to map different arrays and structs to different memories such that parallelism is maximized. Consider the pointer dereference example in Figure 1.

```
1 : int * ptr;
2 : int a[10], b[10], result;
3 : ptr = cond ? a +8 : b+9;
4 : result = *ptr;
```

Figure 1: Pointer dereference example

To synthesize $*ptr$ at line 4, it is necessary to know that the points-to set of *ptr* can potentially include *a* and *b*. Points-to set for a pointer refers to the set of locations that the pointer can potentially point to during the execution of the C++ design. For synthesis, *a* and *b* need to be mapped to the same memory, or logic needs to be generated to access two memories. However, it is not necessary to distinguish between the individual elements of *a* and *b* during behavioral synthesis as it does not impact the memory access logic significantly. To make an impact on the memory layout itself, *all* memory accesses to *a* and *b* will have to be proven orthogonal, and information about a single access is not of much use. In case of verification, the situation is different. In the verification model the pointer dereference of *ptr* has to be modeled as a possible access over all the locations *ptr* could point to. Representing the pointer dereference logic as a disjunction of conjunctions, we have

$$result = ((ptr == a+8) \wedge a[8]) \vee ((ptr == b+9) \wedge b[9])$$

If we didn't distinguish between individual elements of the array, then the expression would contain 20 terms. As a consequence our pointer analysis distinguishes between single elements of the array. Our analysis also distinguishes between individual members of recursive datastructures such as class and struct. Since the points-to set information in our case can be very large, we represent them in terms of pointer ranges instead of the location sets used in [2].

Another difference between behavioral synthesis and our framework is in the handling of loops. Typically in behavioral synthesis, loops are synthesized as state machines. However while constructing a verification model, we cannot model loops as state machines because the verification model has to be cycle accurate to the C++ design. If loops are modeled as state machines, the loop body gets sequentialized, and the verification model then takes more clock cycles to compute the same output compared to the C++ design.

2.2 Pointer Range Analysis

As mentioned in the previous section, we use pointer range analysis to compute points-to sets. For this we have extended the work done in [3] in the following unique ways.

1. We have adapted pointer range analysis to construct optimized verification models from C++. Specifically, the information from pointer ranges are used to construct expressions for pointer reads and writes. All array accesses are handled as pointer accesses as well.
2. We use discrete pointer ranges instead of continuous pointer ranges. For example, given an array $arr[1024]$, we can represent pointer ranges as $a(7,48)$, $a(100,210), a(718,850)$, where $(7,48),(100,210),(718,850)$ are discrete ranges. In continuous range representation, the same information will be approximated as $a(7,850)$. As a result we have higher accuracy while computing the points-to sets.
3. We have augmented the pointer range analysis with the concept of index translation table and function table as discussed later in the Sections 3.6 and 3.7. This allows us to perform pointer range analysis in the presence of polymorphic pointers, dynamic_cast<> and function pointers.

2.3 Pointer Expression Substitution

Our overall verification flow can be compared to the flow described in [4]. However, there are important difference between [4] and our approach in handling the C++ language. In particular, to handle pointers in C in [4], a bottom up back substitution of expressions is done. As a result, each pointer dereference may potentially duplicate a large part of the expressions. Also back substitution of expressions in the presence of complex structures like structs, structs containing pointer arrays etc. becomes very complex, and is not described in [4]. In our approach, the expressions already present in the C++ design are leveraged. As a result no duplication of expressions is introduced by the tool. Also we are able to handle datastructures of arbitrary complexity and polymorphic pointers.

2.4 Static Error Checking for C++

A novel use of semi-formal techniques applied to C++ is described in [5]. In [5] pointer range analysis is used to infer properties about the C++ design. The properties are represented in terms of relationships between the variables of the C++ design. A custom solver is then used to detect possible out-of-range accesses in the design. The similarity to our framework is in the heavy use of pointer range analysis to detect properties of the C++ design. In [5], the range representation can accommodate symbolic bounds as well. Hence the range information in [5] is more precise than our framework where only numerical bounds are supported. However, [5] completely relies on the range information gathered to report errors. Hence the complexity of symbolic bounds for ranges is well justified. In our framework, we rely on the pointer range information to optimize the verification model generated. Formal methods are later used on the constructed model to detect properties. So the model generated in [5] is abstract, but the range analysis used is stronger compared to ours. We construct a precise model of the C++ design, using a weaker range analysis. This seems to work well for C++ designs that are modeling hardware, whereas [5] is targeted toward software. In hardware the interest is in locating precise bugs for relatively smaller models. In software the goal is to scale to large models and be conservative in reporting errors. These constraints lead to further differences in how the C++ design is treated in [5] and in our framework. In [5] loops are not unrolled and functions are not inlined. Instead a depth first search of the dataflow graph is used to explore the C++ design. In our framework we statically unroll all loops and inline all functions as we are interested in the complete functionality of the C++ design.

2.5 Symbolic Simulation

The native interpretation of the C++ datatypes are numeric. However, by converting the interpretation of the datatypes to a symbolic representation, such as BDDs, it is possible to construct a model which represents the functionality of the C++ design for all possible inputs. A very comprehensive overview of this technique is presented in [6]. Our verification model is also functionally precise with respect to the C++ design for all possible inputs. The main difference is that models constructed by symbolic simulation using BDDs are canonical, whereas our verification models are not. As a result, the model constructed by symbolic simulation can be directly used for equivalence checking. In our framework, further effort is required to apply equivalence checking on the models constructed. Thus the verification model construction process and the overall flow is simpler in [6]. However, it is more exposed to the memory and runtime limitations of formal methods as BDDs are constructed for the entire C++ design in one go. By first constructing a memory efficient non-canonical model and deferring the application of formal methods, we have more control over the runtime and memory limitations of formal techniques.

3. CONSTRUCTION OF FINITE STATE MACHINE

In this section we describe the details of constructing a finite state machine from the input C++ design. Specifically we describe our unique contributions towards handling constructs like pointers, dynamic memory allocation, loops, polymorphism, function pointers, virtual methods and floating points.

3.1 Handling Pointers

For the given C++ design, all the variables can be mapped to a memory. This includes scalars, arrays as well as complex datastructures such as classes and structs. Pointers can then be synthesized as indices into this memory. However, in the verification model, each variable access then translates to an access to this combined memory. The complexity of verification is directly related to the number of possible locations accessed for each read or write. The simplistic approach therefore is the worst case scenario where each read in the C++ code gets modeled as a potential read of all the variables. The main focus of our pointer handling framework is to compute very accurate points-to sets so that the logic needed to represent pointer accesses in the verification model is minimized. Details of our pointer synthesis framework :

1. Initialize the range of all inputs to their numeric ranges. For example, an input of unsigned char type will be initialized to the range (0,255) whereas an input of int type will be initialized to the range (-2147483648, 2147483647). Similarly ranges of an array are initialized by initializing the range of each array element. Pointer inputs and outputs are not allowed at the topmost level as they have to be bound to some memory for the C++ design to be considered self contained.
2. The range of any constant at any point is the constant value itself. For example, the constant 10 will be represented by the range (10,10).
3. A separate logical memory is constructed for each variable of fundamental C++ datatype. For example, a variable declared as *int x* will be modeled as a memory named *x* with a single location. A variable declared as *int array*$[128]$ will be modeled as a memory named array with 128 locations.
4. A separate logical memory is constructed for each complex C++ datatype. For example, if the C++ design has a type defined as *class MyClass*, then a logical memory dedicated to the type *MyClass* is constructed. Each location of this memory refers to an instantiation of *MyClass* in the C++ design.
5. Pointers are then assigned ranges based on the logical memories constructed for the fundamental and complex datatypes. This in effect transforms the semantics of the pointers. The pointers in the C++ design refer to memory locations of the virtual memory. However, in our framework they are transformed to indices into the logical memories constructed.
6. At each statement of the C++ design, the operations of the variables are performed as operations on their associated ranges.
7. At conditional statements, the conditions are evaluated based on range arithmetic. Depending on result of the condition evaluation, one or multiple branches are visited. At the end of processing branches of a conditional statement, the values assigned in the different branches are conservatively merged.
8. After range analysis is done for the complete C++ design, the pointer datatypes are converted to integers. All assignments to pointers are converted to appropriate integer index assignments. All reads and writes done using pointers are converted to accesses to the logical memories. The pointers which are now converted to indices into the logical memory point to the desired location for read/write.

For example, consider the C++ code snippet shown in Figure 2.

```
void design_top(unsigned char idx) {
  /* idx is initialized to (0,255) */
  int a[512], b[1024];
  /* a and b are mapped to separate memories */
  int * p1, * p2;
  /* p1 and p2 are mapped to separate memories*/
  p1  = a + idx;
  /* p1 is assigned a(0,255) */
  p2  = b + idx + 10;
  /* p2 is assigned b(10,265) */
  x   = *p1 ;
  /* Potential read over a[0] to a[255] */
  *p2 = x;
  /* Potential write to b[10] to b[265] */
}
```

Figure 2: Pointer range computation

Using the pointer range arithmetic described in [7], we can compute that the read using $p1$ can potentially access $a[0]$ to $a[255]$. As a result, we can model the read as a mux whose select branch selects from $a[0]$ through $a[255]$. Note the size of the mux would have been much larger if a straightforward memory model of 1536 locations was used to model *a* and *b*, or even if read to *a* was modeled as a potential access to 512 locations. The transformed code after pointer range analysis and conversion of pointers to indices is shown in Figure 3.

3.2 Handling Conditional Pointers

To maintain as small a points-to set as possible, each variable of fundamental datatype is mapped to a separate logical memory. However, if a pointer can potentially point to multiple variables which gets decided dynamically, then the logical memories for such variables are combined. To achieve this, during compile time a graph is maintained where each node represents a variable. An edge is added between two nodes if a pointer can potentially point to both the variables. In the end connected components of this graph are computed and each connected component is mapped to a single logical memory. For example, consider the C++ code snippet shown in Figure 4.

Normally, a, b, c and d would be mapped to separate memories and all reads and writes would be disjoint. However, due to the conditional pointer accesses, a, b and c get mapped to a single memory.

```
void design_top(unsigned char idx) {
  int a_mem[512], b_mem[1024];
  /* a_mem and b_mem are the logical memories */
  int p1, p2;
  /* p1 and p2 are converted to integers */
  p1  = 0 + idx;         /* a refers to 0th location of a_mem */
  p2  = 0 + idx + 10;    /* b refers to 0th location of b_mem */
  switch(p1)/* Potential read over a_mem[0-255]*/ {
     case 0   : x = a_mem[0];break;
     :
     case 255 : x = a_mem[255];break;
  }
  switch(p2)/* Potential write to b[10-265] */ {
     case 10  : b_mem[0] = x;break;
     :
     case 265 : b_mem[265] = x;break;
  }
}
```

Figure 3: Pointers to Index Transformation

```
void design_top(bool cond1, bool cond2) {
  int a[4],b[8],c[4],d[16];
  /* Graph initialized to ({a},{b},{c},{d}) */
  int * p1, * p2, * p3;
  p1 = b;
  /* Graph remains ({a},{b},{c},{d}) */
  p2 = cond1 ? a : c;
  /* Graph becomes ({a,c},{b},{d})   */
  p3 = cond2 ? b : a;
  /* Graph becomes ({a,c,b},{d})     */
  int x = *p2;
}
```

Figure 4: Connected Components Computation for Conditional Pointer Assignment : At each stage we depict the state of the graph where nodes within form a connected component.

Lets assume a, b and c are mapped to a common logical memory $abc\_mem[16]$, where a starts at $abc\_mem[0]$, b starts at $abc\_mem[4]$ and c starts at $abc\_mem[12]$. Example in Figure 4 is then transformed to Figure 5.

```
void design_top(bool cond1, bool cond2) {
  int abc_mem[16],d_mem[16];
  int p1,p2,p3;
  p1 = 4;
  /* b is offset 4 into abc_mem */
  p2 = cond1 ? 0 : 12;
  /* a is offset 0 and c is offset 12 */
  p3 = cond2 ? 4 : 0;
  switch(p2)/* p2 was computed abc_mem(0,12)*/ {
    case 0  : x = abc_mem[0];break;
    :
    case 12 : x = abc_mem[12];break;
  }
}
```

Figure 5: Construction of Combined Memory for Connected Components

Note that for $p2$ the range is conservatively computed to be (0,12) when the range (1,11) is actually infeasible. This problem can be alleviated in part using discrete ranges instead of continuous ranges which we discuss next.

3.3 Discrete Range Computation

The use of continuous ranges can sometimes lead to very suboptimal points-to set computations. This is particularly the case in the presence of multi-dimensional arrays. Consider the example in Figure 6. Assuming a row major mapping of *mdim* to its logical memory $mdim\_mem$, $mdim[0][0]$ gets mapped to $mdim\_mem[0]$, $mdim[1][0]$ gets mapped to $mdim\_mem[256]$ etc.

Due to the row major layout of *mdim*, the range computed for $ptr2$ is very suboptimal. If a column major layout is chosen then the suboptimality will shift to $ptr1$. To alleviate the situation, we

```
void design_top(unsigned char idx) {
  int mdim[256][256];/*mapped to mdim_mem[65536]*/
   int * ptr1, * ptr2;
   ptr1 = mdim[128][idx];
   /* ptr1 range is mdim_mem(32768,33024) */
   ptr2 = mdim[idx][128];
   /* ptr2 range is mdim_mem(0,65408) */
}
```

Figure 6: Suboptimal Continuous Ranges

```
// i's range is determined to be (0, 127)
while( i > 0  ) {
  // Loop body
  i--;
}

// Loop unrolling transformation
if(i > 0)  {
  // loop body
  i --;
}
while(i--)  // i's range becomes (-1,126)  {
  // loop body
}
}
```

Figure 7: Loop unrolling

use discrete ranges at selected places. Using discrete ranges, the range of $ptr2$ will be computed as $mdim\_mem(0, 128, ..., 32640)$. Operations on discrete ranges are considerably more complex than the simple continuous ranges. Hence discrete ranges are used only at places where the extra overhead is expected to payoff.

3.4 Handling Dynamic Memory Allocation

Dynamic allocation using $malloc()$ as well as *new* are supported. Each dynamic memory allocation is translated into a static array declaration. The name of the static array is internally generated, while datatype and number of elements are inferred from $malloc()$ or *new* arguments. In case the number of elements is dynamically decided, the maximum of the range computed for number of elements is used. For example, consider a call such as $malloc(N * sizeof(int))$, where range of N is computed to be (0,127). This $malloc$ call is treated as an array declaration of type *int* with 128 locations. In cases where the memory requirement for dynamically allocated memory exceeds a predefined upper limit, the model construction is aborted. The predefined upper limit can be controlled by the user.

3.5 Handling Dynamic Loop Bounds

All loops are unrolled while constructing the verification model. To unroll loops we apply the transformation shown in Figure 7 repeatedly, until the loop condition is evaluated to false by range analysis.

After 128 iterations i's range is (-127,0) for which $i > 0$ test returns false and the loop terminates. Note that normally for all variables defined inside an if-else construct we merge the values conservatively. So for the variable i, we should merge the ranges (0,127) and (-1,126) at the end of the first if-else, which leads to the range (-1, 127). Consequently at the end of 128 iterations, the range conservatively would be (-127, 127) and the loop will never terminate. To solve the situation, we notice that while unrolling the second iteration, we can assume that the first iteration must have been executed. As a result, we consider the value defined inside the if-else branch directly, instead of considering the value obtained by merging the values of i along the different control branches. In cases where a loop does not terminate within a predefined upper limit on the number of iterations, construction of the verification

model is aborted. The predefined upper limit on unrolled iterations can be controlled by the user.

3.6 Handling Polymorphism

As described in Section 3.1, for each pointer in the C++ design, we compute the points-to set at each access point. The points-to sets are then used to construct the verification model. To handle polymorphism, we need the ability to translate the points-to set of a certain type to its base and derived types. To accomplish this we introduce the idea of index translation tables. Conceptually index translation tables are lookup tables that are defined for a given type, called the source type, and its base or derived types called the target type. The size of the lookup table is given by the instances of the source type in the C++ design. Each instance of the source type is associated with a particular entry of the lookup table. The lookup table gives the index corresponding to the target type, if valid, else returns an invalid number to indicate null. Consider example in Figure 8.

```
// (A) : Given type definitions and instances
// of the types

Class Base{};
Class M : Base{};
Class N  : Base{}
M  m1; N   n1; Base b;  M  m2;

// (B) : Each instance of a type is associated
// with an index number via the logical memory.

Base[0] : m1::Base
Base[1] : n1::Base
Base[2] : b
Base[3] : m2::Base
M[0] : m1
M[1] : m2
N[0] : n1

// (C) : The index translation table from Base
// type to M type is shown. On the left the index
// of Base is mentioned, and on the right the corresponding
// translated index of M is shown.

ITT_Base_to_M[0] : 0
ITT_Base_to_M[1] : -invalid-
ITT_Base_to_M[2] : -invalid-
ITT_Base_to_M[3] : 1

// (D) : A simple C++ code using dynamic_cast

Base * bptr;
//..operate on bptr
M * mptr = dynamic_cast<M *>(bptr);

// (E) : dynamic_cast<> are converted to index
// index translation table lookups in the verification
// model constructed.

int bptr;
// operator on bptr
int mptr = ITT_Base_to_M[bptr];
```

Figure 8: Example showing polymorphism handling

There are three types defined, class *Base*, and then derived from it are class *M* and class *N*. A snapshot of the logical memories for *Base*, *M* and *N* is shown in Figure 8(B). The index translation table for *Base* to *M* is shown in Figure 8(C). Note that corresponding to index 1 and 2, the index translation table has invalid entires. This is because *Base*[1] corresponds to *n*1's *Base*, and *Base*[2] refers to *b*. None of these are related to *M*. In Figure 8(D) we show a sample code and in Figure 8(E) the translated code is shown. Note that by indexing into the index translation table, an index to the logical memory corresponding to *Base* is converted to an index into the logical memory for *M*.

```
// (A) :  Sample code using function pointers
int incr(int i) { return i+1; }
int decr(int i) { return i-1; }

int (* fptr) (int i);
fptr = cond ? incr : decr;
y    = *fptr( x );
// (B) : Table which maps functions to integer domain

incr : 0
decr : 1

// (C) : Synthesizing function pointers based to integer ranges

fptr = cond ? 0 : 1;
switch( fptr) {
    case 0 : y = incr ( x);break;
    case 1 : y = decr( x );break;
}
```

Figure 9: Example showing function pointer handling

3.7 Handling Function Pointers

Function pointers are also handled using pointer range analysis. A table of functions present in the C++ design is created. The table maps each function to a unique integer. Pointer ranges for function pointers are then constructed using the table of functions as a reference. Example in Figure 9 illustrates the concept.

3.8 Handling Virtual Methods

C++ compilers implement virtual functions using the concept of virtual method tables [7]. We also keep similar information for each class with virtual methods. Further all methods are mapped to the table of functions described in the last section which allows method accesses to be treated as function pointer access. At each method lookup instance, the virtual table is indexed to determine the correct method pointer. The method pointer is then accessed just like a regular function, with the invoking context set as *this*. The pointer range computed for the invoking expression allows us to trim down the virtual table indexing performed to lookup the correct method.

3.9 Handling Floating Points

Most of the verification backend can handle bit and bitvector datatypes. C++ datatypes such as int, long, char, bool can all be converted to bitvectors. However, floats and doubles cannot be directly converted to bitvectors. As a result floating point operations pose a major obstacle in applying formal techniques to general C++ code. To solve this problem we have adapted the softfloat library [8] to be integrated automatically with a given C++ design. The softfloat library provides a way to model all floating point operations in terms of integer arithmetic. The basic steps are as follows.

- All float and double variables are translated to datatypes provided by the floating point library.
- All operations of float and double are converted to function calls by appropriately looking up the floating point library and hooking up the correct function.
- All floating point constants are recomputed as bitvectors appropriate for the floating point library.
- All typecasts involving floating point are converted to typecasts to the datatypes provided by the floating point library.

This preprocessing removes all floating point datatypes from the input C++ program while maintaining functional equivalence.

4. EXPERIMENTAL RESULTS

In this section we focus on two sets of experiments. The first set of experiments presents results on constructing the verification models in isolation. These experiments are done to ascertain the quality of the verification models constructed. The verification models are dependent on the complexity of the given C++ design. We take the product of runtime and memory of the C++ design as an indication of its complexity. This is compared against the size of the verification model constructed. The size of the verification model is determined from the number of nodes in the verification model and their relative complexity. Also to determine the performance of the model construction process, we present the memory and runtime consumed in constructing the verification models. A brief description of the C++ designs : 1) Armulator is open source ARM instruction simulator 2) SimpleScalar is a processor model from UW Madison 3) Huffman is an adaptive Huffman encoder 4) ADPCM is an adaptive pulse code modulation codec 5) FIR is a floating point model for a 128 tap finite impulse response filter. The results are presented in Table 1.

C++ Designs	Sim Runtime X Sim Memory (secs X MB)	Size of models (1000 gates)	Build time (secs)	Build memory (MB)
Armulator	5323	35672	779	507
SimpleScalar	955	14340	663	264
IDCT	534	7728	79	165
Huffman	467	771	73	216
ADPCM	272	797	25	45
FIR	264	37733	425	168

Table 1: Verification model construction

As it can be seen, sizes of the verification models are roughly correlated to the complexity of C++ designs as defined by the product of simulation time and simulation memory. This is to be expected as the verification model represents an unrolled view of the C++ design. In this context the anomaly between Huffman and ADPCM is hard to explain. Looking deeper reveals that ADPCM does lot more computation, whereas huffman is relatively sparse in computation. The computations do not effect the simulation runtime as much as memory access and conditional statements. Hence the relative complexity of ADPCM comes out lower. However, in the verification model, the complex operations of ADPCM dominate the gate count. This is further confirmed by the fact that the build runtime and memory of ADPCM correlates with the design complexity, but due to the gate count being dominated by large operators, the model size is relatively larger. The extra processing needed for floating point designs is the reason FIR is taking relatively larger time to build compared to its complexity. As we expand each floating point operation to multiple integer operations, the logical gate count is also significantly more for FIR compared to its design complexity.

The second set of experiments are done with the complete equivalence checking flow. We take the C++ designs used in Table 1 and create a buggy copy of the designs. We then use the equivalence checking setup to detect the bug. No clear trends can be concluded from the data due to complex nature of the falsification process and multitude of methods used. Predicting the behavior of the verification backend and drawing meaningful correlations with the C++ design remains a challenging task. This is because it is difficult to predict from the verification model what will be the big obstacles for the verification backend. However, this data does provide an indication of the gross runtimes and memory complexity of the verification process. The results are presented in Table 2.

The final set of experiments is done with the C++ - RTL equivalence checking flow. The results are presented in Table 3. The complexity of the designs are measured in terms of the gate count of the RTL designs. The data is not suitable for drawing general conclusions. However the data gives representative figures for runtime and memory for the size of RTL blocks that we are currently able to handle with ease.

C++ Designs	Falsification Runtime (secs)	Falsification Memory (MB)
Armulator	1860	547
SimpleScalar	1333	298
IDCT	189	192
Huffman	147	354
ADPCM	45	45
FIR	1036	293

Table 2: Verification flow results for C++ vs C++

Designs	RTL gate count (1000 gates)	Runtime (secs)	Memory (MB)	Outcome
Filter	121	41	46	Falsified
Encryption	21	116	231	Proven
Filter2	58	81	245	Proven
Comm	7	990	275	Proven
Comm2	19	2828	1107	Falsified
Video	85	1425	963	Proven

Table 3: Verification flow results for C++ vs RTL

5. CONCLUSION

Sequential equivalence checking of C++ designs is a new concept. Our work extends the frontiers of sequential equivalence checking by making it applicable to real life C++ designs. Analysis of C++ programs and sequential equivalence checking has been well researched and a rich treasure of techniques is present. We have adapted some of the techniques to sequential equivalence checking of C++. However the capacity limitations of formal methods will demand that more such techniques must be adapted or extended, while inventing new ones concurrently. Hopefully this will be an active area of research in the near future.

6. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[2] L. Semeria, Koichi Sato, and G. De Micheli. Synthesis of Hardware Models in C With Pointers and Complex Data Structures. *IEEE Transaction on VLSI*, 9(6), December 2001.

[3] S. H. Yong and S. Horwitz. Pointer Range Analysis. In *Intl. Static Analysis Symposium*, pages 133–148, 2004.

[4] E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Design Automation Conference*, pages 368–371, 2003.

[5] A. Chou Y. Xie and D. Engler. ARCHER: Using Symbolic, Path-senstive Analysis to Detect Memory Access Errors. In *11th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering*, pages 327–336, 2003.

[6] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Intl. Journal of Parallel Programming*, 33(6):645–666, 2005.

[7] Virtual method table. http://en.wikipedia.org/wiki/Virtualtable.

[8] J. Hauser. Softfloat. http://www.jhauser.us/arithmetic/SoftFloat.html.