Preface

Computers have been key subsystems in various complex systems. As computers are adapted into various fields, hardware and software are increasing in size and complexity. It is evident that parallel computing is the way to solve large scale complex information technology problems.

Engineers designing hardware and software are required to verify the system for correctness. As system's size and complexity increases, it is difficult to perform manual system verification. Model checking converts a hardware or software solution into temporal logic and uses solvers to assert on properties of solution. A Bounded Model Checker can verify properties of program/logic within bounded limits. CBMC is a Bounded Model Checker for ANSI-C and C++ programs.

Thesis work is done considering an Ericsson's multicore platform as case study, which uses DSP-C as programming language. DSP-C is a set of language extensions on ISO C programming language. These extensions allow programmers to describe features of Digital Signal Processor (DSP). The work includes extending CBMC to support DSP-C, identifying Bounded Model Checking (BMC) techniques to cope-up with concurrency of Ericsson's multicore Digital Signal Processor (DSP) platform and implementing new features in CBMC to detect issues with Ericsson's parallel software.

Contents

List of Figures

List of Tables

Acknowledgements

This is a master thesis submitted in partial fulfilment of the requirements for degree of Masters of Science in Embedded Systems to Department of Information Technology, Uppsala University, Uppsala, Sweden.

I would like to thank Dr. Philipp Rümmer, Thomas Magnsson and Kenneth Andersson for initiating the work, supporting continuously and providing key insights.

I would like to thank Prof. Parosh Abdulla for reviewing my master thesis.

I express my gratitude to Prof. Daniel Kröening for helping me understand and providing updates of CBMC.

Special thanks to Therese Breinordh and Mats Svensson for taking care of all the administrative formalities at Ericsson.

I would like to thank Uppsala University and Ericsson for providing me all the software and hardware necessary for project.

Sincere thanks to my friends and my family who gave me courage and support throughout the thesis work.

Formal notations

$\wedge$	Concatenation operation
$\vee$	Disjunction operation
$\neg$	Negation operation
$\cap$	Set intersection operation
$\cup$	Set union operation
X^n	Power (X power of n)
$\vdash$	Infers

Chapter 1

Introduction

The complexity of hardware and software is increasing as the years are passing. With increase in complexity, likelihood of errors is much greater. A major goal of software engineering is to enable developers to implement systems which operate reliably despite the complexity. One of the ways to achieve this is by using *formal methods* [12]. Formal methods are mathematically based techniques, tools and languages for describing and verifying the system. These techniques can greatly increase our understanding of a system by revealing incompleteness, ambiguities and inconsistencies that may go undetected otherwise [20].

Single core processor's speed is limited by the physics of semiconductors. High performance computers are being designed using multiple cores to reach the high computation goals. In multicore systems, applications are designed to execute in parallel, and computation speed is achieved by parallel computation. The parallel computation increases complexity of software and hardware. Research industry is working on developing tools and techniques to reduce complexity and detect possible error cases. In this work we have analysed an embedded multicore Digital Signal Processor (DSP) architecture and software, and developed *model checking* techniques. DSPs are processors with special functional blocks to handle digital signals. A digital signal is a sequence of discrete values which represent a physical signal, for example, representing a radio signal or audio signal. Digital signal processing can be enhanced by features like fixed point arithmetic, coprocessors and dedicated registers. The standard C language does not have explicit support to handle these features. Industry and researchers have defined extensions like Embedded-C and DSP-C to add these features to standard C. We have added support for DSP-C in our bounded model checking tool to process DSP-C based programs. We will present multicore, parallel processing, DSP functionality and DSP-C in later chapters.

Model checking is a formal method for verifying logical correctness. Proving logical correctness can be very effective in the development process since testing lacks the coverage [45], peer review is error prone and costly. For example as we can see in Figure 1.1, the function *greatest_common_divisor* can iterate in while loop based on values of x and y, which are dynamic values. There are

$2^N * 2^N$ possible inputs and 2^N outputs, where N is number of bits in int data type. In large software it is impractical to cover all the inputs, outputs and behaviours of each function and module. There are alternative approaches in testing, like code coverage techniques and white-box testing, which can provide some assurance of behaviour but testing cannot prove the correctness.

```
int greatest_common_divisor(int x, int y)
{
      while(x > 0 && y > 0)
      {
           if(x > y)
                x=x-y;
           else
                y=y-x;
      }

      return (x+y);
}
```

Figure 1.1: Function to find greatest common divisor

Verification techniques are being employed extensively in hardware and embedded system development. Since, hardware and embedded systems are designed, developed once and then mass-produced, and bugs in implementation cannot be fixed once produced. Even a single bug in system may lead to recall of all the products. Also, embedded systems have become part of our regular life. For example, microwave oven at home to safety critical systems like power plant controller or flight controllers in aircraft. A single bug in these systems can lead to fatal disasters. Verification techniques can help us to identify possible error cases. In embedded systems, verification can identify problems like checking if array access is within the defined array bound, dangling pointers, arithmetic overflow or underflow and if it is a multicore systems, data races, deadlocks and many other properties can be verified [44, 34, 41].

Model checking *statically analyses* the implementation and asserts on properties of the logic. Figure 1.2 shows the block diagram of model checking. A model checker reads program or circuit logic, converts it into a formula and compares it with the specification. Model checker matches logic and specification, and tells the implementation is incorrect if the logic differs from specification. Model checking is used to detect undesired behaviours of the system.

Software model checking is complex since it operates based on information available statically, without running programs and programs contain code segments controlled by dynamic conditions [18]. Software model checking also has to cope up with loops which are bounded with run-time conditions, for instance, while loop in function *greatest_common_divisor* (Figure 1.1) is bounded by values of x and y. Dynamically bounded programs can be verified using *Bounded*

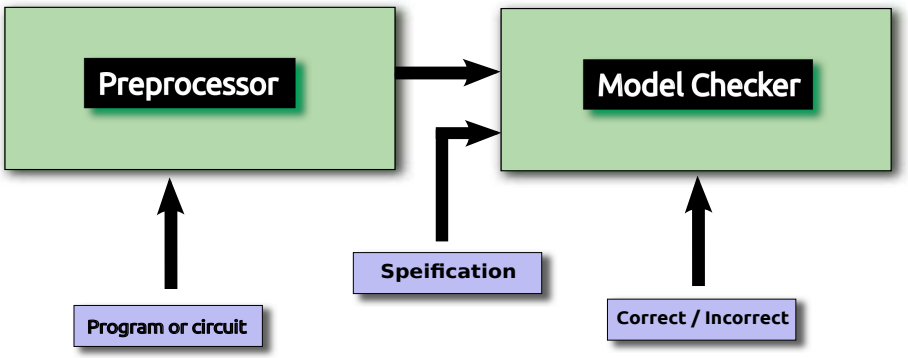

Figure 1.2: Model checking [8]

Model Checking (BMC), which considers a static bound on loops [3]. For instance Figure 1.3 shows *greatest_common_divisor* with loop unrolled. In this example static bound used for loop unrolling is 5. As we can see assert statement at the end is used to verify if bounding limit is not enough. We should note that bounded model checking does not guarantee complete program correctness, since it verifies programs within bounds.

CBMC is a Bounded Model Checking tool which can process C and C++ programs and verify different properties [27, 10, 11]. It converts programs into intermediate forms which are called *goto-programs*. The goto-programs are simplified C and C++ programs, represented in the form of Control Flow Graphs (CFG). In goto-programs, variables are renamed so that each variable is assigned only once, the transformation is called *Static Single Assignment* (SSA) [11]. CBMC also supports pointers, arrays, structures, floating point operations and function pointers. CBMC handles loops by bounding the number of iterations each loop can be executed and unrolling each loop according to the bound. We will present CBMC with more details in later chapters.

Contributions

Thesis work presents a study done to develop a model checking tool for Ericsson's real time DSP multicore platform. The platform uses DSP-C as its programming language. DSP-C extends the ISO C programming language with key features of Digital Signal Processing (DSP) that enable efficient source code compilation [16]. DSP-C adds the following features to ISO C:

- Fixed point arithmetic operations and data types
- Divided memory spaces
- Circular arrays and pointers

We will cover more about DSP-C in later chapters. Ericsson uses contract based programing, which helps large teams working together on same software.

It allows programmers to define the contracts for each module and/or functions. This style of programming provides a framework where module integration is less error-prone since each developer states the requirements for their modules in the contracts [30]. With this thesis we are providing a verifier to check validity of contracts among the function and/or modules.

We developed techniques to handle Ericsson's parallel software running on multicore DSP platform. Major challenge with the parallel/concurrent software verification includes state space explosion due to several control flow paths of parallel programs. Software architecture used in our case study does not pose state-space explosion issue since the software is statically scheduled and software does not share much data between threads and threads run independent of other threads. We also identified some of the platform features which can be verified using model checking and proposed model checking techniques.

Structure of the thesis report

In second chapter we will cover the features of DSP-C, programming model of Ericsson, satisfiability (SAT) solvers, introduce CBMC and discuss about platform specific properties. Third chapter describe the related work in verification. Fourth chapter describe the multicore hardware models and Ericsson's multicore platform. Fourth chapter briefly covers the extensions developed for CBMC to work with Ericsson's software and API stubs to handle platform API calls. In fifth chapter we will discuss about results of model checking, alternative approaches, conclusion. Last chapter is dedicated to propose possible future work.

```
int greatest_common_divisor(int x, int y)
{
     if(x > 0 && y > 0)
     {
          if(x > y)
               x=x-y;
          else
               y=y-x;

          if(x > 0 && y > 0)
          {
               if(x > y)
                    x=x-y;
               else
                    y=y-x;

               if(x > 0 && y > 0)
               {
                    if(x > y)
                         x=x-y;
                    else
                         y=y-x;

                    if(x > 0 && y > 0)
                    {
                         if(x > y)
                              x=x-y;
                         else
                              y=y-x;

                         if(x > 0 && y > 0)
                         {
                              if(x > y)
                                   x=x-y;
                              else
                                   y=y-x;

                              assert(!(x > 0 && y > 0));
                         }
                    }
               }
          }
     }

     return (x+y);
}
```

Figure 1.3: Loop unrolled greatest common divisor function

Chapter 2

Background

This chapter introduces the tools and techniques used in this thesis. We will briefly describe all the feature of DSP-C in first section. Second section concentrates on contract based programming. Third section explains satisfiability (SAT) techniques. Fourth section describes features and architecture of CBMC.

2.1 DSP-C

As the name suggests DSP-C is a programming language extension proposed by a private company, Associated Compiler Experts (ACE). It is an extension to ISO/IEC IS 9899:1900 (ISO C) standard to support the hardware features of Digital Signal Processors (DSP's) [37]. These extensions are proposed to overcome the standard C language's inability to handle divided memory spaces, circular buffers, dedicated register sets, fixed point data-types and fixed point arithmetic [36].

Fixed point

Computing machines like calculators, computers or embedded controllers represent all the numbers in binary. Integers can be directly mapped to finite bit stream of binary digits. Common way to represent fractions is using *floating points* or *fixed point*. Floating point arithmetic supports wider range of values as it has "floating" decimal point. The number is represented using *significant* and *exponent*. [32] presents the various advantages and disadvantage of floating point numbers. Commonly embedded systems do not support floating point values since floating point arithmetic require large logic, needs more computation time and energy [39]. Common alternative is to use fixed point values. Fixed point use fixed number of digits after decimal point. Fixed point values are stored similar to integer values and the decimal position is known since its constant.

Fixed point data type

Programmers can use fixed point data types as easily as any other data types in C language, to describe fixed point arithmetic operations. Explicit support for fixed point types in programming language will allow compiler developers to design fixed point specific optimisations in compilers [36]. It also provides a standardised mechanism to define and use fixed point data types.

DSP-C defines the following fixed point data types:

__fixed types

Signed __fixed and unsigned __fixed types will have a mantissa value. A __fixed object represents values in the range of [-1.0, +1.0]. Number of bits used to store a __fixed is platform specific. For example, a platform specific variant of __fixed type is defined in Table 2.1. Similarly, a platform may define short __fixed to be 16 bits.

Type	**Size in bits**	**Value range**	**Step size (least value greater than 0)**
short __fixed	8	-1.0r to 0.9921875r	0.0078125r
unsigned short __fixed	7	0.0ur to 0.9921875r	0.0078215ur
__fixed	16	-1.0r to 0.99996928..r	0.000030..r
unsigned __fixed	15	0.0r to 0.99996928..ur	0.000030..ur

Table 2.1: A platform specific definition of __fixed types
[NOTE: Succeeding 'r' and 'ur' in above value ranges represents a __fixed type signed and unsigned constants, respectively.]

__accum types

The __accum type is similar to __fixed type with extra 8bits bits to store value before decimal point. For example __accum can store 3.142, but __fixed can only store values between [-1, +1], like 0.142. DSP-C specification [37] defines that "__accum type shall have the same scaling factors as the corresponding __fixed types, with an extension of 8 bits, an __accum value can represent value between [-256.0 to +256.0]". For example, a platform specific variant of __accum type is defined in Table 2.2.

Operations on new data types

DSP-C also defines operations and behaviours of all the operations on data types. It supports all standard C operations on new data types, such as arithmetic operations, logical operations, relational operations and a special qualifier. A special qualifier is __sat, which is only applicable to the __fixed type, makes

Type	**Size in bits**	**Value range**	**Step size (least value greater than 0)**
short __accum	16	-256.0a to 255.9921875a	0.0078125a
unsigned short __fixed	15	0.0ua to 511.9921875ua	0.0078215ua
__fixed	24	-256.0a to 255.99996928..a	0.000030..a
unsigned __fixed	23	0.0r to 511.99996928..ua	0.000030..ua

Table 2.2: A platform specific definition of __accum types
[NOTE: Succeeding 'a' and 'ua' in above value ranges represents a __accum type signed and unsigned constants, respectively.]

a __fixed into __sat a qualified value, which is used during the expression evaluation phase [37]. The sat qualifier adds a saturation operation to expressions. The saturation operation returns same value if the value is less than maximum storable value in __fixed type, otherwise it returns maximum storable value. This operation avoids over flow conditions.

Divided memory spaces

DSP-C allows programmers to provide distributed memory views to compilers. Since memories in DSP's can be physically located in different places, providing divided memory view to developer gives them flexibility to decide on memory location for each variable. This is achieved through memory labelling. When a variable is defined, the label on definition tells compiler which memory will hold a particular variable.

Example

```
__X   int a;
__Y   int b;
```

In above example, variable a and b may be allocated in different memory regions, which can be in different physical memory. For instance, memory label $__X$ will inform compiler that variable will be allocated in memory bank X and $__Y$ will inform that variable will be allocated in memory bank Y.

Dedicated register sets

DPS's normally have register set for dedicated operations. DSP-C provides register labelling to directly access these register set. Programmers can define variables with register labels, similar to memory labels, and force compilers to allocate a variable into particular register.

Circular buffers

DSP-C allows arrays to be defined and used as circular buffers. DSP-C defines new data type (__circ) to make a simple array into a circular buffer. For example in Figure 2.1, "arr[1] = 1;" will copy 1 in array index 1 and "arr[12] = 2;" will copy 2 in index 2.

```
__circ char arr[10];
arr[1] = 1;
arr[12] = 2;
```

Figure 2.1: An example of circular buffer

2.2 Contract programming

The contract programming is a technique, in which the developer states specific requirements for software components. A software component can be function or a module. Contracts define rules on how a component should be used. Contracts contain prerequisites before using the component and define outcomes after using component. A component user has to satisfy all the prerequisites of component being used and agree on possible outcomes, to use the component. For instance contracts of function f can have precondition *pre* and postcondition *post*. Contracts of function f are satisfied if the function f is invoked in state satisfying *pre* and either f does not terminate, or in final state of executing f, the post-condition *post* holds. These contracts help developers to write code under a safety net and components with contracts tend to be less error prone. [30]

```
/**
 * @function: increment pointer
 * precondition (ptr != NULL)
 * postcondition (ptr != MAX_MEMORY_ADDRESS)
 */
int * incrementpointer(int *ptr)
{
    ptr++;

    return ptr;
}
```

Figure 2.2: An example of contract programming

In the example Figure 2.2, precondition checks if the pointer is a NULL

pointer and postcondition checks if the memory address reaches its max value.

2.3 Satisfiability (SAT)

Satisfiability (SAT) has been hot research topic, since SAT has shown high potential in verifying large systems [31]. SAT solvers work using satisfiability procedures in the core for propositional logic [14]. Propositional logic is a predicate in which formula contains Boolean variables, known as atoms, and variables are connected using logical directives like conjunction, disjunction and negation. If z is Boolean variable and, exp_1 and exp_2 are expressions built from Boolean variables, then we can define following formulas.

- z is a Boolean variable and can be evaluated to 0 or 1.
- exp_1 is expression containing Boolean variables.
- $\neg exp_1$ is expression containing negation on exp_1.
- $exp_1 \vee exp_2$ is a disjunctive expression on two expressions.
- $exp_1 \wedge exp_2$ is a conjunctive expression on two expressions.

$(\neg X \vee Y) \wedge (Y \wedge Z))$ from Figure 2.3, is an example for formula constructed from above rules. And the formula can be evaluated to 0 or 1 based the values of all the variables. For example with $X = 1$, $Y = 0$ and $Z = 0$ assignment formula is evaluated to 0, and $X = 0, Y = 1$ and $Z = 1$ assignment formula is evaluated to 1. This example illustrates, variables can be constrained through operators, for instance for formula to be 1, Z must be 1. *Boolean Satisfiability* of a formula is a process of finding an assignment which evaluates it to 1. In this example $Y = 1$ and $Z = 1$ assignment will make the formula to be evaluated to 1 and satisfy it. The formulas which cannot be satisfied with any possible assignment are called unsatisfiable. For example, $(\neg a \vee b) \wedge (a \vee \neg b)$ cannot be satisfied with any assignments and hence unsatisfiable.

SAT problem is NP-Complete [29]. Most SAT solvers use restricted representation of formulas in Conjunctive Normal Formula (CNF). A formula in Conjunctive Normal Form (CNF) is a congestion of clauses. A clause is disjunction of literals. A literal is a Boolean variable, or negation of Boolean variable. For example, $(a \vee b \vee \neg c \vee d) \wedge (a \vee b \vee \neg d \vee e)$, here $(a \vee b \vee \neg d \vee e)$ is a clause with set of variables with or without negation. The approach for finding satisfiability differs in different tools. One of the commonly used approaches is DPLL [13]. In DPLL, given a CNF formula, the algorithm heuristically chooses an unassigned variable and assigns it a value, 0 or 1, this step is known as branching step. Then solver tries to simplify the consequences based on deduction rule. In deduction it tries to deduce if any of clause become 0. If one of the assignments leads 0, the algorithm back tracks since it will not lead to any satisfiability. Once it assigns a combination of values to all the variables which can be 1, the formula is said to be satisfiable.

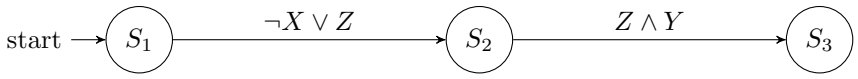

Figure 2.3: An example state-machine for verification

2.4 CBMC

CBMC is an open source Bounded Model Checker for ANSI-C and C++ programs [27]. Bounded model checking is a technique to verify programs within defined bounds. We will discuss more about bounded model checking in next section. CBMC compiles ANSI-C or C++ into goto-programs and verifies properties of the program using bounded model checking techniques. The goto-programs are simplified C and C++ programs, represented in the form of Control Flow Graphs (CFG). The properties includes checking if an assertion is true, array bound limits, dangling pointers, arithmetic overflow/underflow and some other platform specific properties as listed in [27]. Figure 2.4 shows the block diagram of CBMC. Front end compiles source code to intermediate form, called goto-programs. The loops in goto-programs are unrolled.

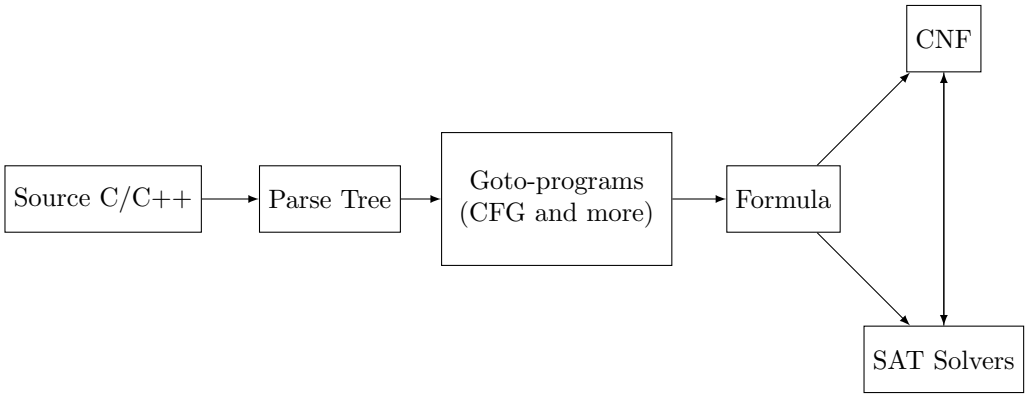

Figure 2.4: Block diagram of CBMC

Loop Unrolling

Loop unrolling, also called as loop unwinding, is process of converting loops into sequential statements. For example:

Example Figure 2.5 can be simply converted as shown in Figure 2.6, which contains sequential statements.

Typically there are also loops which are bounded by run-time conditions, for example in Figure 2.7. C code in Figure 2.7 contains a while loop which terminates when array_b encounters an end of string character (\0) in its index. Static analysis may not provide any information about the contents of arry_b and

```
for(i=0; i<5; i++)
{
    array_a[i] = array_b[i] + 100;
}
```

Figure 2.5: An example of loop with static condition

```
    array_a[0] = array_b[0] + 100;
    array_a[1] = array_b[1] + 100;
    array_a[2] = array_b[2] + 100;
    array_a[3] = array_b[3] + 100;
    array_a[4] = array_b[4] + 100;
```

Figure 2.6: An example of unrolled loop with static condition

it is impossible to know the number of iteration loop will run during execution. Most of the tools use bounded loop unrolling, i.e. if the exit condition for a loop cannot be determined statically, loops are unrolled a maximum of N number of times. Number N can be adjusted according to application. For instance above loop in Figure 2.7, with N set to 5, can be transformed as shown in Figure 2.8.

```
while(array_b[i] != '\0')
{
    array_a[i] = array_b[i];
    i++;
}
```

Figure 2.7: An example of loop with dynamic condition

Assert statement, at last, can be used to check if unrolling was not enough.

Goto-programs

Goto-program is compiled source code, which stores program's information in a structured way. The information includes Control Flow Graph (CFG), data types of the variable, type conversions, library functions and etc.

Variable renaming

Programs also have variables with multiple assignments on same control flow path and it adds complexity on the way we verify programs. To avoid the complexity, variables are renamed whenever new values are assigned. It is known as Static Single Assignment (SSA). This process is done on goto-programs before

```
{
    if(array_b[i + 0] != '\0')
    {
        array_a[i + 0] = array_b[i + 0];
        if(array_b[i + 1] != '\0')
        {
            array_a[i + 1] = array_b[i + 1];
            if(array_b[i + 2] != '\0')
            {
                array_a[i + 2] = array_b[i + 2];
                if(array_b[i + 3] != '\0')
                {
                    array_a[i + 3] = array_b[i + 3];
                    if(array_b[i + 4] != '\0')
                    {
                        array_a[i + 4] = array_b[i + 4];

                        if(array_b[i + 5] != '\0')
                            assert(0);
                    }
                }
            }
        }
    }
}
```

Figure 2.8: An example of unrolled loop with dynamic condition

converting programs into propositional logic. For example, source code shown in Figure 2.9 can be represented as:

$a = 10 \wedge sum = sum + a \wedge sum > MAX\_VALUE$

As we can see in the expression, *sum* is assigned a value and used as a source in the expression. It is not possible to represent such expressions in proposition logic. To avoid it, CBMC converts logic as shown in Figure 2.10.

```
a=10;
...
sum = sum + a;
...
assert(sum > MAX_VALUE)
```

Figure 2.9: An example of Multiple assignments

```
a0 = 10;
...
sum1 = sum0 + a0;
...
assert(sum1 > MX_VALUE);
```

Figure 2.10: Renaming variables

Bit vector flattening

After compiling a source file, we get goto-program. Next step is to verify properties of program and technique used to check these properties is called decision procedure. A decision procedure is a program which terminates with definite answer, true or false, for a decision problem. The decision procedure can decide on control flows based on previous assignments/operation. For example, decision procedure can identify if a control flow can trigger an assertion based on previous assignments and operations.

Standard ways of implementing decision procedure is bit vector flattening followed by a call to a propositional SAT solver. In this process first step is encoding statements from goto-program into bit vectors. Encoding variables and constants to bit vectors is a straight forward task, for example a variable X of size N, can be encoded into bit vectors b of length N. Bit vector operations have to be handled on individual bases. For example, let X, Y and Z be integer variable and a[n], b[n] and c[n] be the bit vectors for each variable respectively. For addition of two bits, we can use a one bit full adder circuit as in Figure 2.11. The circuit will provide us with following formula.

$S_i = (a_i + b_i + C_{in}) mod_2 \longleftrightarrow (a_i \oplus b_i \oplus c_i)$

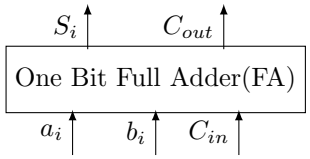

Figure 2.11: One Bit Full Adder (FA)

$C_{out} = (a_i + b_i + C_{in})div_2 \longleftrightarrow (a_i \cdot b_i + a_i \cdot C_{in} + b_i \cdot C_{in})$

Bit flattening is a process of transforming bit vector logic into propositional logic [28]. For example above bit vector logic can be converted to a propositional logic for S_i.

$(a_i \vee b_i \vee \neg C_{out}) \wedge (a \vee \neg b \vee C_{in} \vee \neg C_{out}) \wedge (a \vee \neg b \vee \neg C_{in} \vee C_{out}) \wedge$
$(\neg a \vee b \vee C_{in} \vee \neg C_{out}) \wedge (\neg a \vee b \vee \neg C_{in} \vee C_{out}) \wedge (\neg a \vee \neg b \vee C_{out})$

Similarly we can build carry chain adder for bit vectors, subtractor for subtract operation, bit wise operations etc. The multiplication, division and modulo operations generate large formulas. To handle large expressions and large operations incremental flattening is used [28].

As we know from section 2.3, a proposition logic can be verified using SAT solvers. SAT solver can work on propositional logic of program logic to verify expressions and check different properties of the program logic.

CBMC keywords

Apart from automatically checking properties of program, CBMC also provides set of keywords, which can be used to aide CBMC with more information about program. These keywords can be used for programs instrumentation. The program instrumentation is a procedure change or adds part code to verify some properties of the code.

1. $assert(expr)$ or $\_\_CPROVER\_assert(expr)$ can be used to assert on any condition. It takes a Boolean expression $expr$ as argument. When CBMC encounters one of these keywords, it tries to generate a formula to check assertion failure. Generated formula is verified using SAT-solvers. If the formula is satisfied then assertion fails and CBMC generates error and produces counter-example showing possible trace of error.

2. $\_\_CPROVER\_assume(expr)$ keyword reduces the number of program traces that are considered and allows assume-guarantee reasoning. As $assert$, $\_\_CPROVER\_assume(expr)$ also takes a Boolean expression [10].

Use cases of CBMC

CBMC was used in several case studies, including Bounded Model Checking of Concurrent Programs [35], Equivalence Checking [38], Worst Case Execution Time analysis [43, 24] and many other can be found in [26].

2.5 Verifying properties of thread local and concurrent threads

Model checking is also used to verify specific properties of a system. Some properties are local to a single thread running in the system and some depend on multiple threads running concurrently. The software concurrency in a single core system is introduced by context switches and parallel computing platforms have concurrent execution paths.

Thread local properties include memory access mechanisms, correctness of memory management and pattern of mutex accesses. For example work done in [15] is used to identify DMA race conditions in IBM cell processors. In this case study, DMA race detection is achieved through *program instrumentation*. In program instrumentation, a part of code is added or modified to verify some properties of the logic. For example, to verify array bound, one can add an assertion to check the index value on every array access.

In concurrent programs we can observe more complex properties. For example, concurrent access to shared resources, signalling between the threads and dynamic memory management among threads or race conditions among the threads. Behaviour of these properties depends on the hardware and operating system support. The program instrumentation can be used to verify these properties.

Chapter 3

Related work

Initially logical systems were verified using proof based systems. In proof based verification, system description is represented using a set of formulas γ in a suitable logic and specification is represented using another formula θ. The verification of system is done by finding proof that $\gamma \vdash \theta$. As we can see this process is deductive and usually requires human guidance [22, 1].

The work done by Vardi and Wolper in [40], provided a way of modelling the program specifications into formulas which can be verified automatically. According to this proposal, once expected behaviours and use cases are decided, all the requirements are written into formal specification, which is mathematical description of the system. The formal specifications are written in Linear time Temporal Logic (LTL) and the LTL logic are verified to check the properties of the system. If the system described using LTL behaves as expected, the system is said to be bug free.

Currently we have techniques to convert programs described in high level programming language to mathematical formulas and automated verification technique to verify the properties of the formulas. The section 2.4 describes more details of converting programs to verifiable mathematical formulas and verification using Bounded Model Checking.

Specifications and program are converted into mathematical formulas and the formulas have to be verified for correctness and check for incorrect behaviours. For example consider model checking of state-machine in Figure 2.3. State S_3 can be reached through S_1 and S_2 under the condition $(\neg X \vee Z) \wedge (Z \wedge Y)$. Suppose we want to know if S_3 is reachable under certain conditions, which may violate the specification and is an incorrect behaviour. We need techniques to process the formula $(\neg X \vee Z) \wedge (Z \wedge Y)$ and check if it is satisfiable. Such techniques are called *decision procedures*. Two of the commonly used decision procedures are *Binary Decision Diagrams (BDDs)* and *Satisfiability (SAT)* [25]. In previous chapter we have discussed about satisfiability and next section describes BDD briefly.

"A program verifier uses automated mathematical and logical reasoning to check the consistency of programs with their internal and external specifica-

tions" [23]. Hardware and software verification considers checking the correctness of functionality and finding undesired behaviours in the designed logic. There are several stages and ways in which a system can be verified. During the design process a system specification is developed and a mathematical model can be implemented to verify if the properties of specified system are as expected. The implemented logic can be converted into mathematical logic and this mathematics logic can be verified for its correctness. There are already tools which can work with system specifications like UML and automatons. For example BCC and UPPAAL. Also there are tools which can work with implementation done in programming languages like C, C++, Verilog or VHDL. For example, CBMC, SatAbs and VCEGAR.

Binary Decision Diagrams (BDDs)

Graph based verification techniques like *Binary Decision Diagrams (BDDs)*, as described in [5]. The BDDs are proven to be very effective in verifying binary logic [4]. In this approach the model is described in-terms of a Directed Acyclic Graph (DAG) consisting of decision nodes and terminal nodes.

Although BDDs are effective in solving verification problem, as the number of variables/nodes increase the size of BDDs increases exponentially and it is not practical to use BDDs, since verification process will be too slow and too memory consuming [7]. There have been attempts to develop techniques which can avoid the exponential growth, for instance work shown in [6, 2, 33] and work by Bryant in [5] showed that ordering variables will increase the efficiency of the algorithm. But despite all the efforts state explosion has been a major hurdle in applying BDD-based model checking to large and complex systems [9].

Chapter 4

Multicore Hardware Model

The computers are used in diverse applications. Applications may provide unique challenge on the way computers process data. For example some application may have one algorithm to be run large amount of data and some others may require large amount of instructions to be executed on small amount of data. These requirements have defined the architecture of computers.

In this chapter we briefly present the Flynn's taxonomy, discuss about multicore memory models and data sharing concepts.

Model and architecture of hardware used for this thesis are intellectual property of Ericsson. This chapter only presents common multicore architectures.

4.1 Computing platforms

The computing platforms are generally classified based on Flynn's taxonomy [17]. The classification is based on number of data and instruction streams. The classification is as follows:

1. Single Instruction Stream-Single Data Stream (SISD)

   A computing unit with single instruction stream and single data stream. For example: single core micro-controllers and micro-processors.

2. Single Instruction Stream-Multiple Data Stream (SIMD)

   Multiple computing units process same instruction on different data stream. For example: Graphical Processing Units (GPU).

3. Multiple Instruction Stream-Single Data Stream (MISD)

   Multiple computing units process different instruction on same data. For example fault tolerant systems run different algorithms on same data and analyse the result from both the algorithm.

4. Multiple Instruction Stream-Multiple Data Stream (MIMD).

Multiple instructions streams work on different data. For example, general purpose parallel computers.

Parallel computing makes use of multiple computational units to process data at same time (in parallel) [21]. The architecture of parallel computers can be SIMD, MISD, MIMD or *heterogeneous architecture*, which is a combination of these architectures.

4.2 Multicore Memory models

The memory model defines the organisation and access mechanisms of computer memory. The memory models are designed to address application specific requirements. The memory models can be classified into Uniformed Memory Access (UMA), Non-Uniform Memory Access, Cache Only Memory Access and Scratchpad Random Memory Access (SRAM).

UMA

In UMA architecture, all the processors share a common main memory and any processor's memory access time to any of the memory region of main memory is independent [21].

NUMA

In NUMA architecture, all the processors share a common main memory and access time to a memory region depends on the address space of main memory it is accessing [21].

COMA

In COMA architecture, processors do not have a main memory, instead the processors are interconnected and support caches. The data is accessed through caches and *cache coherency* protocol is used.[19, 21]

SRAM

In SRAM architecture, all processors support memory blocks supported controlled by programs, called scratchpad memories, as replacement for cache [21].

4.3 Data sharing

The parallel programs running in different cores or processors can share data to produce results. Data sharing can be done through sharing memory or through message transfer. In multicore systems, cores are physically near and it is less time consuming to share memory. Any of the above memory models can be used

for data sharing. To maintain correctness of data shared by parallel programs, programmers have option of using mutex and *coherency protocols* to keep the data consistent among different cores.

Chapter 5

Implementation

In this chapter we will present implementation part of the thesis. First section describes about DSP-C support implemented in CBMC, second section is dedicated to contract programming, third section talks about memory overlay handling and last section presents the support for parallel programs in multicore platform.

Note that, software we were working with was property of Ericsson. We cannot publish real examples, to avoid intellectual property rights violation. We are using toy programs as examples to present the concepts.

5.1 DSP-C support

As we presented DSP-C in section 2.1, DSP-C extends ISO C with new data types and operations. Also we looked at framework of CBMC in section 2.4. CBMC processes source code, produces goto-programs, goto-programs are converted propositional logic and verified using SAT-solver. To begin with we needed to add support in parser for parsing new data types, constants, operations and memory labels. Then parsed data formats have to be stored in goto-programs in a structured way to be processed during propositional logic conversion. The propositional logic conversion module has to be updated for handling new structures in goto-programs.

We updated new grammar to handle new data types, constant types and memory labels. With working parser to parse the new type, we added logic to handle automatic type conversions between fixed arithmetic to any of ISO supported basic data types, like int, float or character.

CBMC has been updated recently to convert fixed types to propositional logic. It works by remembering total number of bits to store a fixed point and bits for fraction in it. We developed a logic in which we can feed these numbers dynamically, during conversion. With this modification, we can handle multiple types of fixed point, type conversions and operations. The current implementation can easily work with addition, subtraction and multiplication

of fixed point type.

The modified CBMC tool can process features of DSP-C and some of the simple test runs are been presented in Appendix A.

As we understood from section 2.1 memory labels provide information about storage location of each variable and pointer's pointing location. In current implementation, we store all the memory labels in goto-programs as property of variables. In future one can use memory labels to verify the properties related to memory.

5.2 Contract verification

We studied the use cases and advantages of contract programming in section 2.2. Our goals were to implement a mechanism to verify, if a postcondition can be proven false based on precondition and verify if the caller's contract violate the callee's contracts or callee's contracts can violate caller's contacts. For instance Figure 5.1 shows a generic pseudo code for function caller and callee with contracts. As we can see there can be two kinds of contract violations possible in $function\_callee$. First, postcondition of a function can fail because of its own preconditions. For example, *pre_cond* can be $state == X$ and *post_cond* can be $state == Y$, and there can be feasible path with state to be X, but does not have assignment to state variable to be Y. This will violate the postcondition based on precondition. Second, precondition of callee, $precondition(pre\_callee)$, can fail because of precondition of caller $precondition(pre\_caller)$. For example, callee can have precondition with $state == X$ and caller with $state == X$, and a feasible path from precondition of caller to precondition of callee with assignment $state = Y$ can violate the contract of callee.

The implementation proved to be much simpler since CBMC already provides keywords like $\_\_CPROVER\_assume$ and $\_\_CPROVER\_assert$. To verify contract violation within a function, we had to replace all the callee preconditions with $\_\_\_CPROVER\_assert$ and caller initiator with $\_\_CPROVER\_assume$. It can be simply done by defining appropriate macro to replace during preprocessing stage of CBMC. When CBMC starts processing preconditions and postcondition, it assumes on initial function entry and asserts on every callee's precondition.

For second type of contract verification, we replaced preconditions with $\_\_CPROVER\_assume$ and postcondition with $\_\_CPROVER\_assert$. When CBMC starts processing a function it assumes all the preconditions and starts processing statements between precondition and postcondition. If assumption together with following operations violates the postcondition, violations are caught in $\_\_CPROVER\_assert$.

In example shown in Figure 5.2, *open_door* function is written to automatically open the door. It has preconditions for checking if the lock is not on and door is not opened already. Postcondition checking the door is open once operations are performed. The CBMC tries to check contracts between caller and calee, it will make sure that caller satisfies the requirements before it calls the

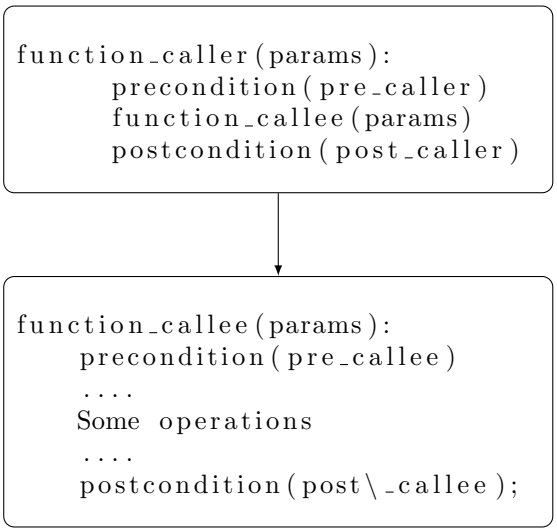

Figure 5.1: Function calls in contract programming

```
/**
 * @function: open_door
 *            This function automatically opens the door.
 * precondition (door_locked == false && door_open == false)
 * postcondition (door_open == true)
 * invariant(door_locked)
 * returns result: 0 if success, -1 if failure
 */
int open_door(BOOL door_locked, BOOL door_open)
{
    precondition (door_locked == false && door_open == false)

    /* Operations to automatically open the door */
    ...
    ...

    door_open = true;

    postcondition (door_open == true)
}
```

Figure 5.2: Example of contract programming

function. Also when contacts are used to verify functionality of a function, we can easily find any issues with postcondition based on preconditions. Appendix B presents an example run.

5.3 Platform specific libraries and macro support

In this section we will briefly summarise the platform related library support we implemented.

Memory overlays

In some applications part of data is never reused or written back to permanent storage. Also, when we look at embedded systems, in most cases executing instructions are always kept in read only memory and whenever necessary data is brought to instruction memory, cache or scratchpad. Memory overlay is a process in which part of memory is overwritten without worrying about previous contents of it.

```
int random_generator()
{
    ...
}

int print_number(int num)
{
    ...
}

int main()
{
   int number;
   ...

   number = random_generator();
   print_number(number);
   ...
}
```

Figure 5.3: An example of multiple function calls

In example shown in Figure 5.3, main generates a random number and prints it before going further with other processing. In case, programmer knows she or he is not going to use *random_generator* again, the same address can be taken

by *print_number*. Possibly one can also rewrite the program as shown in figure Figure 5.4.

```
int random_generator()
{
    ...
}

int print_number(int num)
{
    ...
}

int main()
{
   int number;
   ...
   number = overlay_and_call(memory_region_x,
                             random_generator());
   overlay_and_call(memory_region_x,
                    print_number,
                    number);
   ...
}
```

Figure 5.4: An example of memory overlay

As we can see function *overlay_and_call* takes a pointer to memory region, function pointer and variable number of argument to be supplied to the called function. When the function executed, instructions of function, whose pointer is supplied as parameter, are brought to specific region of the memory mentioned in the argument. This avoids any memory write backs and provides faster processing. Also this reduces the amount of memory required for applications [42].

We added support to handle memory overlays. For data overlay we treat each new memory overlay as new memory block allocation and old block as removal. The new instruction overlay and call to those instructions is treated as simple function call. This allows CBMC to handle memory overlays in simplest manner. In future work one can add support to verify the dangling pointers due to overlays.

Support for parallel programs

Parallel programs have several control flow paths and produce large formulas during model checking. The platform we analysed did not pose threat of state

space explosion since processes are scheduled to reduce the concurrent processes accessing same data. Tasks created by a process are designed to operate on independent data block and process waits until all the task finish their job. Once all the tasks finish their operation, process collects the results. To handle this kind of parallelism we modified CBMC libraries to treat each task creation as function calls. When multiple tasks are created once, we call the task one by one and order of call does not affect result since each task works on independent data blocks.

Internal macros

The home grown compiler for the platform supports various internal macros. These macros are specific to compiler and platform. With these macros programmers can perform platform specific tasks. For example, Open-MP like macro can parallelise a loop over different hardware threads.

Chapter 6

Discussion and Conclusion

During this thesis work we extend CBMC to handle features of DSP-C, provided a model to verify contract programming and proposed verification techniques to detect issues in Ericsson multicore DSP processor's parallel code.

In following sections we briefly discuss results of model checking performed on Ericsson's software. To begin with we used extended CBMC to check logical correctness of the code. In second step we performed contract verification. At last with collective knowledge of Ericsson's platform architecture and model checking on software, we proposed techniques to verify some more properties of the software.

Model checking

Test setup started with understanding Ericsson's build system, which is capable of handling large code base being maintained by large number of developers distributed across globe. The build system is complex to cater its requirements. The code base includes millions of lines of code and hundreds of developers working on it. For simplicity we started with modules which had little library dependency. Even with these modules we did have to port most of the Ericsson specific Real Time OS APIs. Also we should note that these modules were already tested and deployed in products.

With the modified CBMC, API stubbing/porting and understanding of build system, we were able process libraries of Ericsson. During the test run we observed a lot of false positives on assertions since these assertions were written to handle run time errors. This we mainly attribute to the fact that existing code is not written to provide information to static analysers. As we understood in chapter chapter 2 and chapter 5, static analysers work by looking at information available in the existing logic without knowing run time information. To make the best of any static analyser one has to provide run time information statically. This information can be provided as part of header in each module or function, commented information for analysers or through contracts. The code we were working had contracts and next section provides more details on contract based

verification.

Contract verification

Most of the Ericsson code has contracts on function implementation. As we discussed in section 5.2, we stubbed preconditions and postconditions with CBMC keywords. During Ericsson's code analysis we realised that contracts used in the code are run time guards, which is a way of pessimistically checking incoming parameters at run time. These conditions check values of each parameter when a function is called and return values are checked before returning. Contracts are not providing enough information to CBMC to verify the properties of each function.

In verification, contracts are expected to be defined on behaviour of the function and also provide actual limits of parameter(s). Preconditions for the function can include limits on parameters and limits on global states. Postconditions can include affected states and results being returned. The affected states can be either part of parameters or global variables.

Identifying properties of thread local and concurrent threads

Apart from verifying logic properties of software, like arithmetic errors or pointer analysis, model checking can be used verify specific properties of the system. For example race conditions within a thread or between threads, memory handling or mutex handling.

Thread local race conditions are seen in system where the processor core and processor peripheral access processor common memory asynchronously. For example a case study in [15] shows IBM cell processor with possible race condition between processor core and DMA controller. Ericsson platform uses a strong memory consistency model. During analysis we concluded that the DSP cores do not have thread local race conditions. Bounded model checking can verify some of thread local properties. For example:

- **Correctness of memory allocation and deallocation**

  The dynamic memory management APIs can be instrumented to detect multiple deallocation of same memory address in a single thread.

- **Correctness of handling mutex variables**

  The mutex handling APIs can be instrumented to detect multiple locking/unlocking on same mutex variable.

As we discussed in earlier chapters, concurrent software have several control flow paths and produce large formulas during model checking. The platform we analysed is designed to operate on incoming signals. Typically multiple signal handlers run to completion on parallel cores. Majority of signal handler operations are independent of other signal handlers in the system. In some cases signal handlers share some resources. The signal handlers also take advantage

of heterogeneous architecture and spawn threads to slave cores and accelerate the computation. Bounded model checking can be used to verify following properties of the platform.

- **Verifying race conditions among the cores**

  In some cases signal handlers share resources, like shared data structures or shared IO devices. Bounded model checking can be used to identify possible race conditions.

- **Memory management**

  Signal handlers make use of slave cores and transfer some of the processing to slave cores. The slave cores are capable of accessing shared memory regions, allocate or deallocate memory. Bounded model checking can be used to verify correctness of the memory usage by multiple slave cores.

- **Correctness of using mutex variables**

  The shared memory access is done within protected area. We can instrument mutex APIs to detect possible unprotected access to shared memory regions.

Chapter 7

Future Work

In current implementation we keep memory labels as variable properties in goto-programs. In future we can use these labels in verifying memory related operations.

As we discussed in previous sections, contracts defined in the code are not providing enough information to verifier. Ericsson's code can be carefully reviewed again and one can add contracts which provide enough information about functionality. Also, contracts can be defined effectively, through top down development approach. In top down development approach, product architecture is designed, architecture is divided into modules and each module will have specific functionality. This approach clearly identifies system states being used by each module. It is easier to identify possible preconditions, postcondition and invariants.

In previous chapter we have discussed various properties we identified in the system and proposed possible implementation techniques. All the properties can be verified by extending CBMC or instrumenting the software.

Bibliography

[1] K.R. Apt. Ten years of hoare's logic: A survey - part i. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[2] F. Balarin and A. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification*, pages 29–40. Springer, 1993.

[3] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[4] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.

[5] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[6] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10 20 states and beyond, 1990.

[7] E. Clarke. Model checking. In *Foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.

[8] E. Clarke. The birth of model checking. *25 Years of Model Checking*, pages 1–26, 2008.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[10] E. Clarke and D. Kroening. Ansi-c bounded model checker user manual. Technical report, Technical report, School of Computer Science, Carnegie Mellon University, 2006.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[12] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[14] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[15] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of dma races using model checking and k-induction. *Form. Methods Syst. Des.*, 39(1):83–113, August 2011.

[16] DSP-C. An extension to iso c. `http://www.dsp-c.org/`, May 2012.

[17] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[18] P. Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[19] E. Hagersten, A. Landin, and S. Haridi. Ddm-a cache-only memory architecture. *Computer*, 25(9):44–54, 1992.

[20] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.

[21] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Pub, 2011.

[22] C. A. R. Hoare. Communicating sequential processes, 2004.

[23] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50:2003, 2003.

[24] S. Kim, H.D. Patel, and S.A. Edwards. Using a model checker to determine worst-case execution time. Technical report, Citeseer, 2009.

[25] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view*. Springer-Verlag New York Inc, 2008.

[26] Daniel Kroening. The cbmc applications. `http://www.cprover.org/cbmc/applications.shtml`, May 2012.

[27] Daniel Kroening. The cbmc homepage. `http://www.cprover.org/cbmc/`, May 2012.

[28] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, 2008.

[29] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[30] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[31] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[32] M.L. Overton. *Numerical computing with IEEE floating point arithmetic*. Siam, 2001.

[33] C. Pixley, S.W. Jeong, and G.D. Hachtel. Exact calculation of synchronizing sequences based on binary decision diagrams. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(8):1024–1034, 1994.

[34] H. Post and W. Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[35] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *In Computer-Aided Verification (CAV), LNCS 3576*, pages 82–97. Springer, 2005.

[36] M. Roodzant. Real-time dsp applications benefit from high-level language compilation. *REAL TIME MAGAZINE*, page 40, 1999.

[37] DSP-C Speification. An extension to iso c. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n854.pdf`, May 2012.

[38] M. Staats and M. Heimdahl. Partial translation verification for untrusted code-generators. *Formal Methods and Software Engineering*, pages 226–237, 2008.

[39] V. Tiwari and M. Tien-Chien Lee. Power analysis of a 32-bit embedded microcontroller. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 141–148. IEEE, 1995.

[40] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[41] N. Vasudevan and S.A. Edwards. Static deadlock detection for the shim concurrent language. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 49–58. IEEE, 2008.

[42] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109. ACM, 2004.

[43] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based timing analysis. *Leveraging Applications of Formal Methods, Verification and Validation*, pages 430–444, 2009.

[44] F. Werner and D. Faragó. Correctness of sensor network applications by software bounded model checking. *Formal Methods for Industrial Critical Systems*, pages 115–131, 2010.

[45] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.