General comments:
* do better proof reading, language still needs to be improved. some mistakes are marked, but not all
* be more precise; when you introduce terms or notions, give real definitions. do not use abbreviations that were not introduced. do you not use technical terms/notions that were not introduced, or for which you did not give a reference

# Model checking of multicore software using CBMC

Ashok Kelur
`ashok.kelur.1807@student.uu.se`
`ashoksarf@gmail.com`

July 10, 2012

## Abstract

The model checking converts a hardware or software solution into the 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.

The master thesis work includes extending CBMC to support DSP-C, identifying Bounded Model Checking(BMC) techniques to cope-up with concurrency of a Ericsson's DSP multicore platform and implementing new features in CBMC to detect issues with Ericsson's parallel software. We also developed techniques to verify contracts used in Ericsson's software.

Update, give more details, explain acronyms, etc.

# Acknowledgments

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

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

I would like to thank Therese Breinordh and Mats Svensson for taking care of all the administrative formalities at Ericsson.

I would like to thank my parents, Vijaykumar and Vijaya, for supporting me in all of my initiatives and my siblings, Kavita and Ningappa, for love and support.

At last, I would like to thank all my friends for providing helpful suggestions whenever in need.

# Contents

# List of Figures

# List of Tables

# Formal notations

| | | | |
|---|---|---|---|
| | $\vee$ | Concatenation operation | Disjunction |
| Conjunction | $\wedge$ | Disjunction operation | |
| | $\neg$ | Negation operation | |
| | $\cap$ | Set intersection operation | |
| | $\cup$ | Set union operation | |
| ? | | Set negation | |
| | $X^n$ | Power (X power of n) | |

you give rather too many technical details about BMC here. concentrate on the problem at hand, the overall philosophy/methodology for solving it, and the results/outcomes of the project

# Chapter 1

# Introduction

The complexity of hardware and software is increasing as the years are passing. With the increase in complexity, the likelihood of errors is much greater. A major goal of software engineering is to enable developer's to implement systems which operate reliably despite the complexity. One of the ways to achieve this is by using *formal methods* [13]. 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 [21].

say more about the specific topic of the thesis (DSP, multicore, concurrency, etc)

*Model checking* is a formal method for verifying logical correctness. Proving logical correctness could be very effective in development process since testing lacks the coverage [43], 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 the values of x and y, which are dynamic values. There could be $2^N * 2^N$ possible inputs and $2^N$ outputs, where N is number of bits in int data type. In large software it might be impractical to cover all the inputs, outputs and behaviors 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 behavior but testing might not be able to prove the correctness.

can?

be more specific (in general; avoid words like could/would/might)

The model checking *statically analyzes* the implementation and asserts on the properties of the logic. Figure Figure 1.2 shows the block diagram of model checking. A model checker reads program or circuit logic, converts it into a formula and compare it with the specification. The model checker returns true if the logic matches the specification and false if the logic differs from specification. The model checking can also be used to detect undesired behaviors of the system. For example consider state-machine in Figure 1.3, model checking tool would preprocess expressions and suggest if any of the error states are reachable. For instance reaching state $S_{n4}$ could lead to divide by zero exception.

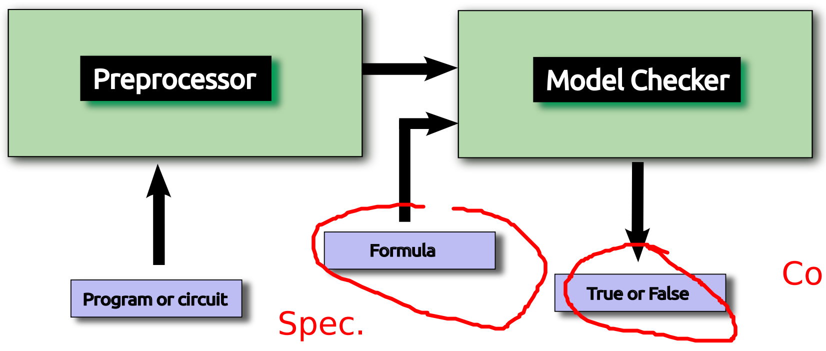
not "also", that's the very core of verification

The software model checking is complex since it operates based on information available statically, without running the programs and programs contain

```
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



Correct/Incorrect

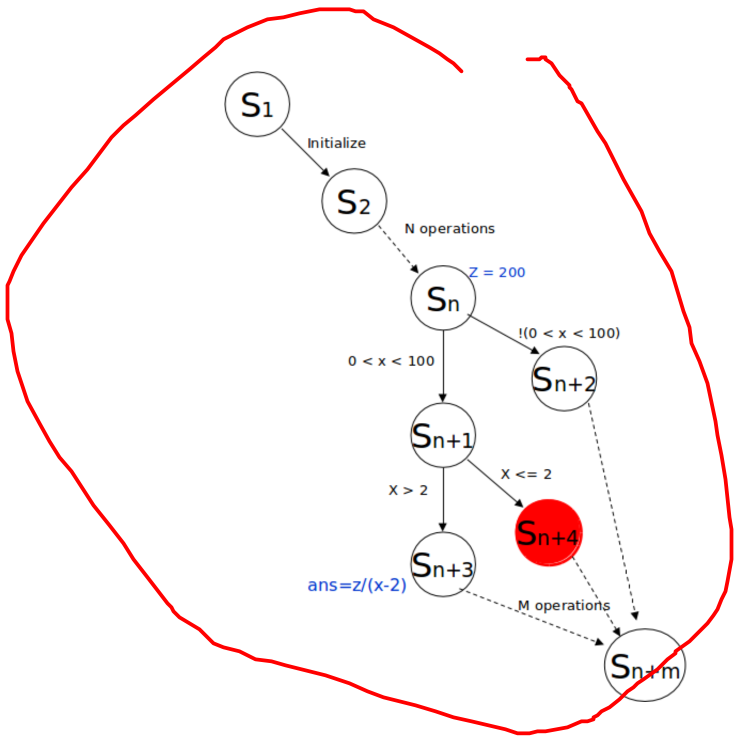
Figure 1.2: Model checking [8]

code segments controlled by dynamic conditions [19]. The 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* is bounded by values of x and y. The dynamically bounded programs are verified using *Bounded Model Checking (BMC)*, which considers a static bound based on human inputs during verification ~~or incremental verification~~ [3]. In incremental verification a program is verified with say N as bound and if no error is found then bound is increased to N+1 and verification continues until a bug is found or process reaches a maximum bound value.

this is just the particular approach we chose, not everyone is doing that

this is not incremental verification, just bounded model checking

*CBMC* is a Bounded Model Checking tool which can process C and C++ programs and verify different properties [28, 10, 12]. It converts the programs into intermediate forms which are called as *goto-programs*. The goto-programs are simplified C and C++ programs, represented in the form of Control Flow Graphs(CFG). In goto-programs, the variables are renamed so that each variable is assigned only once, the transformation is called *Static Single Assignment*



if you want to show this diagram, you have to provide (much) more explanation. what do the nodes represent (control states?), what do the guards on transitions means, etc.

Figure 1.3: State machine example

(SSA) [12]. CBMC also supports pointers, arrays, structures, floating point operations and function pointers. CBMC handles loops by bounding number of iterations each loop can be executed and unrolling each loop according to the bound.

## Contributions

The 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. The DSP-C extends the ISO C programming language with key features of Digital Signal Processing(DSP) that enable efficient source code compilation [17]. The DSP-C adds the following features to ISO C:

say were more details about this are given (later chapter)

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

Ericsson uses contract based programing, which allows large teams distributed across geography to work together on same software. It allows programmers to define the contracts, which can be simple inputs and outputs, for

contracts are not inputs/outputs, they are specifications composed of logical formulae, which state properties about inputs/outputs/state/etc what do you mean by this?

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[31]. With this thesis we are providing a verifier to check validity of contracts among the function and/or modules by *over approximating* the contracts.

We developed techniques to handle Ericsson's parallel software running on multicore DSP platform. Major challenge with 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, mostly run independent of other threads. We also identified some of the features of platform 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, the history of verification techniques, introduce CBMC and discuss about platform specific properties. The third chapter describes the multicore hardware models and Ericsson's multicore platform. The third 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. The last chapter is dedicated to propose possible future work.

# 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 the first section. The second section concentrates on the contract based programming. The Third section explains history of verification techniques and current trend. The fourth section describe features and architectures of CBMC.

## 2.1 DSP-C

As the name suggests DSP-C is a programming language extension proposed by 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 registers sets, fixed point data-types and fixed point arithmetic [36].

### Fixed point

In mathematics a real number contains all the rational numbers such as integer and fraction. The computing machines like calculators, computers or embedded controllers represent all the numbers in binary. The integers can be directly mapped to finite bit stream of binary digits. The common way to represent fractions is using *Floating points* or *Fixed point*. The floating point supports wider range of values as it has "floating" decimal point. The number is represent using *significant* and *exponent*. The [33] 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]. The common alternative is to use fixed point values. The fixed point use fixed number of digits after decimal point. The fixed point values are stored similar to integer values and the decimal position is known since its constant.

"floating-point arithmetic"

a real number is only a real number. you mean probably: the set of real numbers is a superset of the set of integers, and the set of rationals

### Fixed point data type

The programmers could use fixed point data types as easily as any other data types in C language, to describe fixed point arithmetic operations. The explicit support for fixed point types in programming language will allow compiler developers to design fixed point specific optimizations in compilers [36]. It also provides a standardized mechanism to define and use fixed point data types.

The DSP-C describes the following fixed point data types:

#### __fixed types

defines

The signed __fixed and unsigned __fixed types will have a mantissa value. A __fixed object represents values in the range of [-1.0, +1.0]. The 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** unit? | **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 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. The 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

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

| **Type** | **Size** | **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.]

type, makes a __fixed into __sat a qualified value, which is used during the expression evaluation phase [37]. } more details!

## Divided memory spaces

The DSP-C allows programmers to provide distributed memory views to compilers. Since memories in DSP's could 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 labeling. When a variable is defined, label on the definition tells compiler which memory will hold that variable.

**Example**

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

In above example, variable a and b may be allocated in different memory regions, which could be in different physical memory. For instance, memory label __X would suggested that variable should be allocated on memory bank near to processor X and __Y would suggest that a variable should be allocated memory bank near processor Y.

what is a "suggestion" in this context?

## Dedicated register sets

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

say that this aspect is not considered further in the thesis (unless it is)

## Circular buffers

The DSP-C allows arrays to be defined and used as circular buffers. The 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;” would place 1 in array index 1 and “arr[12] = 2;” would place 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

give a proper definition: what are the components of a contract, and what is their semantics.

The contract programming is a technique, in which the developer states specific requirements for the software components. These requirements could contain precondition, post-condition and invariants. As the names suggest, each of these requirements does exactly the same. Preconditions define states/conditions before invoking a function, postcondition define states/conditions after invoking the component and invariants are the states which does not change while processing the component. These contracts help developers to write code under a safety net and components with contracts tend to be less error prone.

ref?

for instance: a contract for a function f is a pair (pre, post), consisting of a pre-condition pre, and a post-condition post. f satisfies the contract (pre, post) if, whenever f is invoked in a state satisfying pre, either f does not terminate, or in the final state of executing f the post-condition post holds.

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

why do pre/post-cond. occur twice?

Figure 2.2: An example of contract programming

In the example Figure 2.2, precondition checks if the pointer is a NULL pointer and post condition checks if the memory address reaches its max value.

people tend to disagree about the precise meaning of a contract, and you have to decide (and define) what you consider as correct. for instance: is it legal to call a function in a state that does not satisfy the pre-condition of the contract of the function? what if the function has many contracts, and only the pre-cond. of some of them are satisfied?

## 2.3 Verification techniques

"A program verifier uses automated mathematical and logical reasoning to check the consistency of programs with their internal and external specifications" [24]. The Hardware and software verification considers checking the correctness of functionality and finding undesired behaviors 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.

Initially logical systems were verified using proof based systems. In proof based verification, system description is represented using a set of formulas $\gamma$ in a suitable logic and specification is represented using another formula $\theta$. 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 [23, 1].

Later in 1981 model checking techniques as shown in [11], proved that model checking could lead to algorithmic and automatic verification process. Model checking is also much simpler since it models a hardware or software description into constrained model with finite states and verifies the models instead of implementation, which hides the complexities of descriptive languages.

be more specific. what is a "constrained model"? why does it have finite state space? (e.g., model in Fig. 1.3 could have infinite state space, due to the variables ranging over infinite domains). also, those "early" forms of model checking are very different from BMC, as you are using it

Vardi?

The work done by Verdi and Wolper in [40], provided a way of modeling the logic into formulas which can be verified automatically. According to this proposal, once expected behaviors and use cases are decided, all the requirements are written into formal specification, which is mathematical description of the system. Usually 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.

the term "logic" is dangerous, it has various meanings. you mean "design"? or "program"?

Currently we have techniques to convert programs described in high level programming language to mathematical formulas. The section 2.4 describes more details of converting programs to verifiable mathematical formulas in CBMC.

The specifications and program are converted into mathematical formulas and the formulas have to be verified for correctness and/or check for incorrect behavior. For example consider model checking of state-machine in Figure 2.3. The state $S_3$ could 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 a incorrect behavior. We need techniques to process the formula $(\neg X \vee Z) \wedge (Z \wedge Y)$ and check if it is possible under the conditions. Such techniques are called as *decision procedures*. Two of the commonly used decision procedures are *Binary Decision Diagrams*

"possible" -> "satisfiable"

under which conditions?

this is a very specific use. you don't write your DSP specifications in LTL, do you?

you only use SAT later
on, not SMT.
SMT is more of a framework, not a specific
decision procedure

*(BDDs)* and *Satisfieability Module Theories (SMT)* [26].

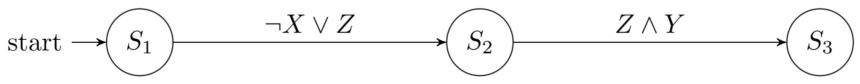


Figure 2.3: An example state-machine for verification

don't spend
too much time
on this topic; it
is not very
relevant for BMC

### Binary Decision Diagrams (BDDs)

they are still popular

Graph based verification techniques like *Binary Decision Diagrams(BDDs)*, as described in [5] were also popular until now. 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. The terminal nodes can be either 0 or 1 and the decision nodes, with out-degree 2, are nodes combining the terminal 0's or 1's or a combination decision nodes. The left edge going out from the decision node represents case when a variable is 0 and right edge represents the case for 1. Figure 2.4 shows a BDD for state machine in Figure 2.3, containing decision nodes $X$, $Y$ and $Z$. The terminal nodes containing 0 or 1 based on the evaluation of the formula $(\neg X \vee Z) \wedge (Z \wedge Y)$. The reach-ability analysis on BDD-DAG provides facilities to check if a particular behaviors is possible in the system. For example we can check if $S_3$ is reachable if $Z$ is 0 and it is not possible since all the left edges from Z reach a 0 terminal node. There are various tools like ABCD, BCC and JINC which can be used to verify programs using BDD.

tree-shaped BDDs
are a very special
case, this might
be misleading

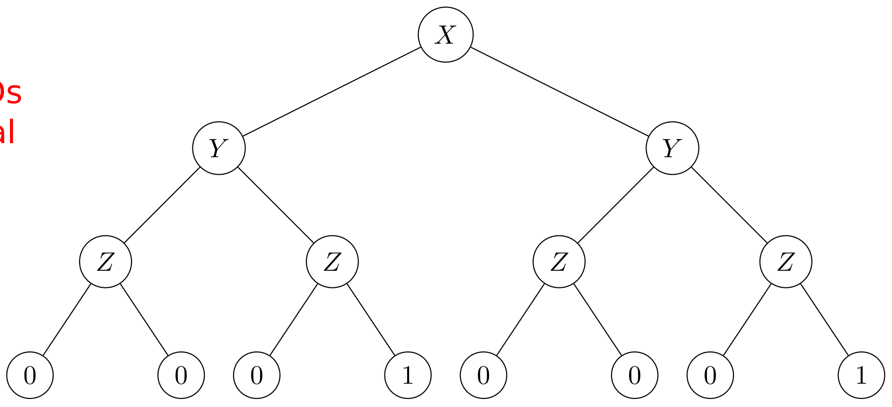


Figure 2.4: Binary Decision Diagram (BDD)

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, 34] 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].

## Satisfieability Modulo Theories (SMT)

this section is more about SAT, not SMT

Satisfieability Modulo Theories(SMT) have been hot research topic, since SMT has shown high potential in verifying large systems [32]. The SMT solvers work using on satisfieability procedures in the core for propositional logic [15]. Propositional logic are predicate logic in which the formulas contain Boolean variables, known as atoms, and variables are connected using logical directives like conjunction, disjunction and negation. ~~For example,~~ $z$ is Boolean variable and, $exp_1$ and $exp_2$ are expressions built from Boolean variables, then we could define following formulas.

there are no functions is prop. logic, so it is better to not use the term here

- $z$ is a Boolean variable and can be evaluated to 0 or 1.
- ~~$exp_1$ is function containing Boolean variables and can be evaluated to 0 or 1.~~
- $\neg exp_1$ is function containing negation on function ~~and can be evaluated to 0 or 1~~.
- $exp_1 \vee exp_2$ is a conjunctive function on two functions ~~and can be evaluated~~ to 0 or 1. "conjunction"
- $exp_1 \wedge exp_2$ is a disjunctive function on two functions ~~and can be evaluated~~ to 0 or 1. "disjunction"

The $(\neg X \vee Y) \wedge (Y \wedge Z))$ 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 $X = 1$, $Y = 0$ and $Z = 0$ would evaluated it to 0, and $X = 0, Y = 1$ and $Z = 1$ will evaluated it 1. This example illustrates how variables have to be constrained through operators, for instance for formula to be 1, $Z$ must be 1. *Boolean Satisfieability* of a formula is a process of finding a assignment which evaluates it to 1. In this example $Y = 1$ and $Z = 1$ would make the formula 1 and satisfy it. The formulas which can not be satisfied with any possible assignment are called unsatisfiable. For example, $(\neg a \vee b) \wedge (a \vee \neg b)$ can not be satisfied with any assignments and hence unsatisfiable.

abbrev. not introduced

The SAT problem is NP-Complete [30]. NP-complete formulas have exponential worst case execution time. Most SAT solvers use restricted representation of formulas in Conjunctive Normal Formula(CNF). A CNF is disjunction of set of clauses, a clause is concatenation of set of variables with or without negation. For example, $(a \vee b \vee \neg c \vee d) \wedge (a \vee b \vee \neg d \vee e \vee)$, here $(a \vee b \vee \neg d \vee e \vee)$ is a clause with set of variables with or without negation. The approach for finding satisfieability differ in different tools. One of the commonly used approach is

if you can show this, you will get the Turing award

Use more precise language.
"A formula in conjunctive normal form is a conjunction of clauses.
A clause is a disjunction of literals. A literal is a boolean variable, or the negation of a boolean variable."

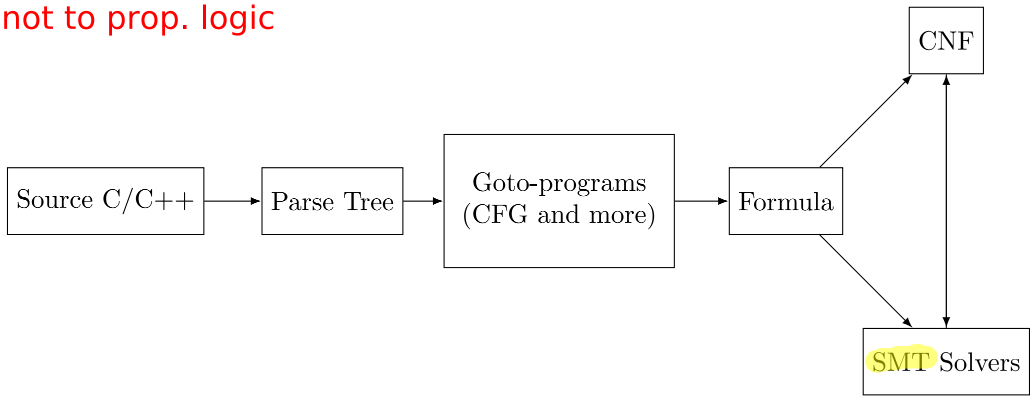
DPLL [14]. In DPLL, given a CNF formula, the algorithm heuristically chooses an unassigned variable and assigns it a value, 0 or 1, this step is known as branching step. Then solver tries to deduce the consequences based on deduction rule. In deduction it tries to deduce if any of clause become 0. If one of the assignment leads 0, the algorithm back tracks since it will not lead to any satisfieability. Once it assigns a combination of values to all the variables which can be 1, the formula is said to be satisfiable. ~~Also there may not be any satisfiable assignment, in which case the formula is unsatisfiable.~~

the whole process is deductive. this particular step is rather called simplification, or propagation

## 2.4 CBMC

The CBMC is a open source tool for compiling C and C++ programs to goto-programs and checking properties of the goto-programs. The goto-programs are simplified C and C++ programs, represented in the form of Control Flow Graphs(CFG). The properties includes checking if a assertion is true, array bound limits, dangling pointers, arithmetic overflow/underflow and some other platform specific properties as listed in [28]. The Figure 2.5 shows the block diagram of CBMC. Front end compiles source code to intermediate form, called goto-programs. The loops in goto-programs are unrolled since, loops can not be converted to formulas.

they can, but not to prop. logic



SAT

Figure 2.5: Block diagram of CBMC

### Loop Unrolling

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

The example Figure 2.6 can be simply converted as shown in Figure 2.7, which contains sequential statements.

Typically there are also loops which are bounded by run-time conditions, for example in Figure 2.8.

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

Figure 2.6: 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 2.7: An example of unrolled loop with static condition

The C code in Figure 2.8 contains a while loop which terminates when array_b encounters a end of string character (\0) in its index. The static analysis may not provide any information about the contents of arry_b and 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 can not be determined statically, loops are unrolled a maximum of N number of times. The number N can be adjusted according to application. For instance above loop in Figure 2.8, with N set to 5, can be transformed as shown in Figure 2.9.

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

### Goto-programs

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

### Variable renaming

be more specific

The programs also have variable with multiple assignments on same control flow path and it adds complexity on the way we verify the programs. To avoid the complexity variables are renamed whenever new values are assigned, its called as Static Single Assignment (SSA). This process is done on goto-programs before converting programs into propositional logic. For example, the source code shown in Figure 2.10 can be translated as below.

this is not a correct translation, so you should not say "can be translated"

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

As we can see it is tricky to handle it in propositional logic since a variable is assigned a value and reused in the formula. To avoid it the CBMC converts logic as shown in Figure 2.11.

use correct logical notation.
so far, you have only introduced prop. logic,
not any logic providing arithmetic

figure should be closer to this page

besides variable renaming, it would be important to show how assertions are translated

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

Figure 2.8: An example of loop with dynamic condition

### Bit vector flattening

decision proc. is a very general term. here, we would rather speak of program checking, or sth. similar

After compiling a source file, we get programs in goto-program. The next step is to check assertions or any other properties 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 could tell if the control flow can trigger a 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, could 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 could imagine a one bit full adder circuit as in Figure 2.12. The circuit will provide us with following formula.

$S_i = (a_i + b_i + C_{in})mod_2 \longleftrightarrow (a_i \oplus b_i \oplus c_i)$
$C_{out} = (a_i + b_i + C_{in})div_2 \longleftrightarrow (a_i \cdot b_i + a_i \cdot C_{in} + b_i \cdot C_{in})$

those are already on the bit level, not bit-vectors

Bit flattening is a process of transforming bit vector logic into propositional logic [29]. 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 [29].

As we know from section 2.3 proposition logic can be verified using SAT solvers. With the SAT solver we can verify the reach-ability, side-effects to/of each variable/expression and assert on the expressions. ???

### CBMC APIs

this is not an API, rather keywords

Apart from automatically checking properties of program, CBMC also provides set of APIs, which can be used to aide CBMC with more information about

```
{
    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 2.9: An example of unrolled loop with dynamic condition

program. These APIs can be also be used programs instrumentation. The program instrumentation is a procedure change or add 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 APIs it tries to generate formula for reach-ability and *exprn*, the generated formula is verified using SAT-solvers. If the formula is satisfied then assertion holds and CBMC generates error and produces counter-example showing possible trace of error.

why reachability?

not right term

2. *__CPROVER_assume*(*expr*) API 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].

also, the translation is such that correct programs lead to unsatisfiable SAT problems, and incorrect programs to satisfiable ones!

CBMC will give an error msg. when an assertion does /not/ hold!

```
a=30;;
...
sum = sum + a;
...
assert(sum > MAX_VALUE)
```

Figure 2.10: An example of Multiple assignments

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

Figure 2.11: Renaming variables

**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 [42, 25] and many other can be found in [27].

## 2.5 Verifying properties of thread local and concurrent threads

The model checking is also used to verify specific properties of a systems. 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.

The thread local properties include memory access mechanisms, correctness of memory management and pattern of mutex accesses. For example work done in [16] is used to identify DMA race conditions in IBM cell processors. The DMA race detection is been achieved through *program instrumentation*. In program

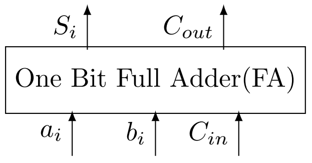


Figure 2.12: One Bit Full Adder (FA)

instrumentation, a part of code is added/modified to verify some properties of the logic. For example, to verify array bound, one could just add a assertion to check the index value on every array access.

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

it would be good to tell a bit more about instrumentation of programs

# Chapter 3

# Multicore Hardware Model

As we know computers are tools process application data and computers are used in diverse applications. The applications may provide unique challenge on the way computers process the 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, data sharing concepts and introduce to the Ericsson's DSP multicore platform.

## 3.1 Computing platforms

The computing platforms are generally classified based on Flynn's taxonomy [18]. 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 single computing unit with single instruction stream and one data stream. For example 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 analyze 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 process data at same time (in parallel) [22]. The architecture of parallel computers could be SIMD, MISD, MIMD or *heterogeneous computer* with combination of these classification.

## 3.2 Multicore Memory models

The memory model defines the organization and access mechanisms of computer memory. The memory models are designed to address application specific requirements. The memory models can 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 [22].

### NUMA

In NUMA architecture, all the processors share a common main memory and access time to a memory region depends on the which address space of main memory it is accessing [22].

### 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.[20, 22]

### SRAM

SPM?

In SRAM architecture, all processors support memory blocks supported controlled by programs, called scratchpad memories, as replacement for cache [22].

## 3.3 Data sharing

The parallel programs running in different cores or processors would have to share data to produce results. The data sharing could 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 could be used for data sharing. To maintain the correctness of the data shared by parallel programs, programmers have option of using mutex and *coherency protocols* to keep the data consistent among different cores.

## 3.4 Ericsson hardware platform

The platform used for this thesis is a home grown ASIC processor from Ericsson. The platform is designed for *LTE* base stations. The is a long range wireless communication technology with high speed data access facilities for mobile phones.

The LTE has large data and instruction parallelism, which has influenced the architecture of the platform. The platform contains multiple cores arranged in heterogeneous architecture with scratchpad memory. Figure 3.1 shows the block diagram of the platform. The platform contains master cores and slave cores with local scratchpad memory. The cores also share a common scratchpad memory. The master cores are designed to handle incoming radio signals. The master core fetches the incoming signal data through peripheral interface. The slave cores are utilized as *hardware accelerators* by the master cores. The master cores creates tasks to process a received signal and posts it on the job queue, which keeps track of all the unfinished jobs. Depending on the priority of each task, job queue runs each task to completion.

not really hardware accelerators, the accelerator cores are still programmed in software

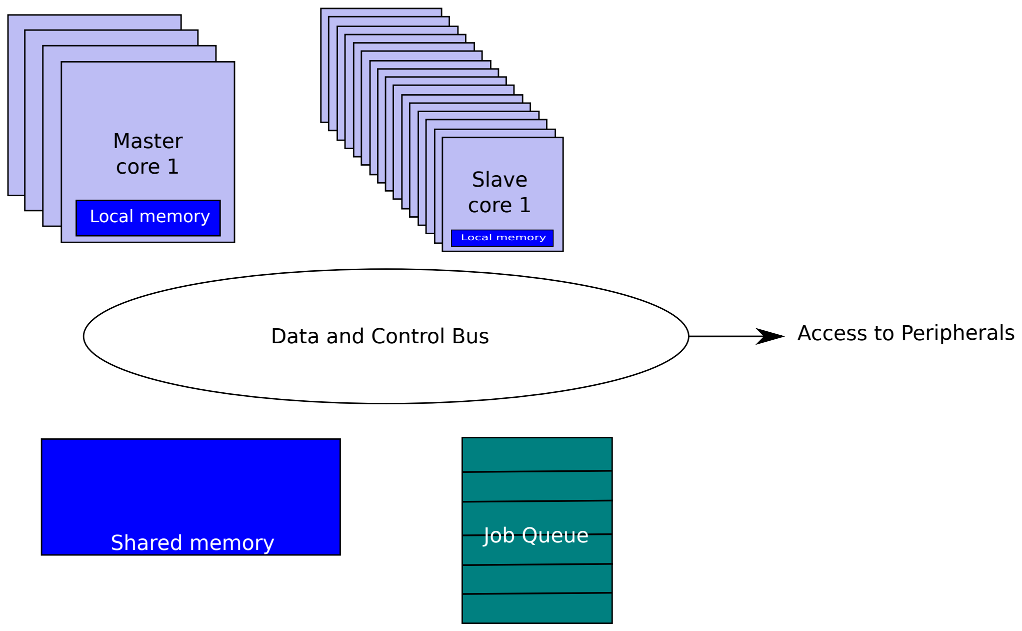


Figure 3.1: Ericsson's DSP multicore platform platform block diagram

# Chapter 4

# Implementation

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

Note that, software we were working with was property of Ericsson and we could not publish real examples, due to Intellectual property rights violation. We are using hypothetical examples to present the ideas.

not due to violation, avoiding violation

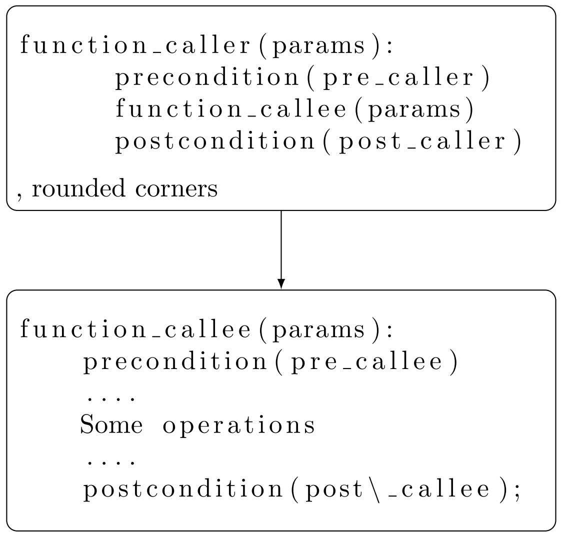
## 4.1 DSP-C support

As we can see in the section 2.1, DSP-C extends ISO C with new data types and operations. Also we looked at the framework of CBMC in section 2.4. As we could see CBMC processes source code, produces goto-programs, the 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 the 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 the new grammar to handle the new data types, constant types and memory labels. With the working parser to parse the new type, we added logic to handle automatic type conversions between fixed arithmetic to any of the ISO supported basic data types, like int, float or character. This logic is necessary since one could implement programs which work with mixed types.

what are mixed types?

The CBMC has been updated recently to convert fixed types to propositional logic. It works by remembering the number of bits for fraction and number for integer. We developed a logic in which we can feed these numbers dynamically, during conversion. By this we could handle multiple types of fixed point conversions and operations. The current implementation could easily work with



```
function_caller(params):
        precondition(pre_caller)
        function_callee(params)
        postcondition(post_caller)
```

, rounded corners

```
function_callee(params):
     precondition(pre_callee)
     ....
     Some operations
     ....
     postcondition(post\_callee);
```

Figure 4.1: Function calls in contract programming

addition, subtraction and multiplication of fixed point type.

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

summarise experience with Ericsson code

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 the variable. In future one could use these information to verify the properties related to memory.

## 4.2 Contract verification

goals?

We studied the use cases and advantages of contract programming in section 2.2. Our tasks were to implement a mechanism in which we could verify if the post condition could hold based on precondition and verify if the caller's contract would violate the callee's contracts or callee's contracts could violate the caller's contacts. For instance Figure 4.1 shows a generic pseudo code for function caller and callee with contracts. As we can see there could be two kinds of contract violations possible in *function_callee*. First, precondition of callee, *precondition(pre_callee)*, might not hold because of precondition of caller *precondition(pre_caller)*. It should also be possible to find a contract violation after N number of nested calls. Second, it might be possible to statically determine if the postcondition of a function is violated based on its own preconditions.

?

this should be much more detailed, and more formal. connect to the definition of contracts given earlier

The implementation of mechanism proved to be much simpler since CBMC already provides APIs like *__CPROVER_assume* and *__CPROVER_assert*.

an important question here: do you verify programs modularly (each function individually), or do you verify a program as a whole?

To verify first contract violation we had to replace all the callee preconditions with ___*CPROVER_assert* and the caller initiator with __*CPROVER_assume*. It could be simply done by defining appropriate macro to replace during preprocessing stage of CBMC. When the CBMC starts processing the preconditions and postcondition, it assumes on initial function entry and asserts on every callee's precondition.

say exactly what this is

For second type of contract verification, we replaced preconditions with __*CPROVER_assume* and post condition 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 violate the postcondition, violations are caught in __*CPROVER_assert*.

```
/**
 * @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 4.2: Example of contract programming

The example shown in Figure 4.2, *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, post condition checking door would be open once the operations are performed and one could also monitor the invariants. When the CBMC tries to check contracts between caller and calee, it would make sure that caller satisfies the requirements before it calls the function. Also when the contacts are used to verify the functionality of a function, we could easily find any issues with post condition and invariants based on preconditions. Appending B presents some example runs.

which?

give a summary how this worked on Ericsson's code

## 4.3 Memory overlays

In some applications some 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 4.3: An example of multiple function calls

In example shown in Figure 4.3, main generates a random number and prints it before going further with other processing. In case if programmer know he is not going to use *random_generator* again, the same address could be taken by *print_number*. Possibly one could also rewrite the program as shown in figure Figure 4.4.

explain in more detail

or she

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 the 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 provide faster processing. Also this reduces the amount of memory required for the applications [41].

In the tool we added support to handle memory overlays. For data overlay we treat each new memory overlay as new memory block allocation and old

```
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 4.4: An example of memory overlay

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 could add support to verify the dangling pointers due to overlays.

more details.
say what dangling pointers are, and what exactly you want to verify

## 4.4 Support for parallel programs

The parallel programs have several control flow paths and produce large formulas during model checking. The platform we analyzed did not pose threat of state state space explosion since processes are scheduled to reduce the concurrent processes accessing same data. The tasks created by a processes are designed to operate on independent data block and process waits until all the task finish their job. Once all the task finish their operation, the 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 the call does not affect the result since each task works on independent data blocks.

to be continued ...

Chapter 5

# Discussion and conclusion

During this thesis work we extend CBMC to handle features of DSP-C, provided a model to verify contract programming and based on hardware model of Ericsson multicore DSP processor, proposed solutions to detect issues in 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

The 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. As one could realize the requirements for such build system, it is complex to cater its requirements. The code base includes about a million lines of code and hundreds of developers. For simplicity we started with small libraries which had little or no library dependency. Even with these libraries we did have to port most of the Ericsson specific Real Time OS APIs. Also we should note that these libraries 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 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 analyzers. As we understood in chapter chapter 2 and chapter 4, static analyzers work by looking at information available in the existing logic without knowing run time information. To make the best of any static analyzer one have to provide run time information statically. This information can be provided as part of header in each module or function,

be more specific.
what information
did you have to add?

commented information for analyzers or through contracts. The code we were working had contracts next section provides more information on contract based verification.

## Contract verification

Most of the Ericsson code have contracts on function implementation. As we discussed in section 4.2, we stubbed the precondition and postcondition with CBMC APIs. During Ericsson's code analysis we realized that contracts used in the code are run time guards, which is a way of pessimistically checking incoming parameters at run time. These conditions would check the values of each parameter when a function is called and a run-time check for the return values before returning. The contracts were not providing enough information to CBMC to verify the properties of each function.

In verification, contracts are expected to be defined on the behavior of the function and also provide actual limits of the parameter. The preconditions for the function could include limits on parameters and limits on global states. The post conditions could include affected states and results being returned. The affected states could 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 could be used verify specific properties of the system. For example race conditions within a thread or between threads, memory handling or mutex handling.

The platform uses a strong memory consistency model. During the analysis we concluded that the DSP cores do not have thread local race conditions. The bounded model checking can verify some of the thread local properties. For example:

define what this is

- **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 produces large formulas during model checking. The platform we analyzed is designed to operate on incoming signals. Typically multiple signals run to completion on parallel cores. Majority of signal handler operations are independent of other signals in the system. In some cases signal handlers share some resources. The signal handlers also take advantage of heterogeneous architecture and spawn threads to hardware accelerators. The bounded model checking could be used to verify following properties of the platform.

- **Verifying race conditions among the cores**

  In some cases the signal handlers share resources, like shared data structures or shared IO devices. The bounded model checking can be used to *over approximate* the access to shared resources and identify possible race conditions.

- **Memory management**

  The signal handlers make use of hardware accelerators and transfers some of the processing to hardware accelerators. The hardware accelerators capable of accessing shared memory regions may access, allocate or deallocate memory. The bounded model checking can be used to verify correctness of the memory usage by multiple hardware accelerators.

- **Correctness of using mutex variables**

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

# Chapter 6

# Future Work

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

As we discussed in previous sections, contracts defined in the code would not provide enough information to verifier. Ericsson's code could 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, a 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] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] 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.

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

[34] 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.

[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] 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.

[42] 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.

[43] 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.

# Appendix A

# Examples of new data type and verification results

Below program presents a example of __fixed type.

```
int main() {
    short __fixed a = 0.001r;
   __fixed b = 0.01r;

   if(a + b > 0.01r)
      assert(0);
}
```

As we can see from above program assertion would hold, same has been detected using CBCM on static analysis and given a trace of it.

```
file ./fixed_type.c: Parsing
Converting
Type-checking fixed_type
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 14 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 with simplifier
99 variables, 146 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
```

```
Runtime decision procedure: 0.002s
Building error trace

Counterexample:

State 2 file <built-in> line 26 thread 0
----------------------------------------------------
  __CPROVER_deallocated=NULL (00000000000000000000000000000000)

State 3 file <built-in> line 27 thread 0
----------------------------------------------------
  __CPROVER_malloc_object=NULL (00000000000000000000000000000000)

State 4 file <built-in> line 28 thread 0
----------------------------------------------------
  __CPROVER_malloc_size=0 (00000000000000000000000000000000)

State 5 file <built-in> line 29 thread 0
----------------------------------------------------
  __CPROVER_malloc_is_new_array=FALSE (0)

State 6 file <built-in> line 38 thread 0
----------------------------------------------------
  __CPROVER_rounding_mode=0 (00000000000000000000000000000000)

State 8 file ./fixed_type.c line 3 function main thread 0
----------------------------------------------------
  a=0 (0000000000000000)

State 9 file ./fixed_type.c line 3 function main thread 0
----------------------------------------------------
  a=0.001007080078125 (0000000000100001)

State 10 file ./fixed_type.c line 4 function main thread 0
----------------------------------------------------
  b=0r (00000000000000000000000000000000)

State 11 file ./fixed_type.c line 4 function main thread 0
----------------------------------------------------
  b=0.00999999977648258209228515625r (00000001010001111010111000010100)

Violated property:
  file ./fixed_type.c line 7 function main
  assertion
  (_Bool)0
```

Below program presents a example of __accum type.

```
int main() {
   short __accum a = 0.001a;
  __accum b = 1.01a;

  if(a + b > 1.01a)
     assert(0);
}
```

As we can see from above program assertion would hold, same has been detected using CBCM on static analysis and given a trace of it.

```
file ./fixed_type.c: Parsing
Converting
Type-checking fixed_type
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 14 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 with simplifier
131 variables, 194 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Runtime decision procedure: 0.001s
Building error trace

Counterexample:

State 2 file <built-in> line 26 thread 0
----------------------------------------------------
  __CPROVER_deallocated=NULL (00000000000000000000000000000000)

State 3 file <built-in> line 27 thread 0
----------------------------------------------------
  __CPROVER_malloc_object=NULL (00000000000000000000000000000000)

State 4 file <built-in> line 28 thread 0
----------------------------------------------------
  __CPROVER_malloc_size=0 (00000000000000000000000000000000)
```

```
State 5 file <built-in> line 29 thread 0
----------------------------------------------------
  __CPROVER_malloc_is_new_array=FALSE (0)

State 6 file <built-in> line 38 thread 0
----------------------------------------------------
  __CPROVER_rounding_mode=0 (00000000000000000000000000000000)

State 8 file ./fixed_type.c line 3 function main thread 0
----------------------------------------------------
  a=0 (000000000000000000000000)

State 9 file ./fixed_type.c line 3 function main thread 0
----------------------------------------------------
  a=0.001007080078125 (000000000000000000100001)

State 10 file ./fixed_type.c line 4 function main thread 0
----------------------------------------------------
  b=0a (0000000000000000000000000000000000000000)

State 11 file ./fixed_type.c line 4 function main thread 0
----------------------------------------------------
  b=1.00999999977648258209228515625a (000000001000000010100011110101110000010

Violated property:
  file ./fixed_type.c line 7 function main
  assertion
  (_Bool)0

VERIFICATION FAILED
Error[10]: while processing ./fixed_type.c
```

# Appendix B

# Contract verification example runs

Consider a faulty implementation of open door where the door may not be opened even though the preconditions are met.

```
/**
 * @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 */

    if(door_locked==true &&  door_open ==false)
         door_open=true;
    else
         door_open=false;

    postcondition (door_open == true);
}
```

The CBMC verifies for the postcondtion and suggests the possible states which could lead failure.

```
file contract.cpp: Parsing
Converting
```

```
Type-checking contract
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 37 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 with simplifier
13 variables, 23 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Runtime decision procedure: 0s
Building error trace

Counterexample:

State 3 file <built-in> line 28 thread 0
----------------------------------------------------
  __CPROVER_deallocated=NULL (00000000000000000000000000000000)

State 4 file <built-in> line 29 thread 0
----------------------------------------------------
  __CPROVER_malloc_object=NULL (00000000000000000000000000000000)

State 5 file <built-in> line 250 thread 0
----------------------------------------------------
  __CPROVER_architecture_int_width=32 (00000000000000000000000000100000)

State 6 file <built-in> line 251 thread 0
----------------------------------------------------
  __CPROVER_architecture_long_int_width=32 (0000000000000000000000000010000

State 7 file <built-in> line 252 thread 0
----------------------------------------------------
  __CPROVER_architecture_char_width=8 (00000000000000000000000000001000)

State 8 file <built-in> line 253 thread 0
----------------------------------------------------
  __CPROVER_architecture_short_int_width=16 (000000000000000000000000000100

State 9 file <built-in> line 254 thread 0
----------------------------------------------------
```

```
  __CPROVER_architecture_long_long_int_width=64 (00000000000000000000000001000000)

State 10 file <built-in> line 255 thread 0
----------------------------------------------------
  __CPROVER_architecture_pointer_width=32 (00000000000000000000000000100000)

State 11 file <built-in> line 256 thread 0
----------------------------------------------------
  __CPROVER_architecture_char_is_unsigned=0 (00000000000000000000000000000000)

State 12 file <built-in> line 257 thread 0
----------------------------------------------------
  __CPROVER_architecture_word_size=32 (00000000000000000000000000100000)

State 13 file <built-in> line 258 thread 0
----------------------------------------------------
  __CPROVER_architecture_fixed_for_float=0 (00000000000000000000000000000000)

State 14 file <built-in> line 259 thread 0
----------------------------------------------------
  __CPROVER_architecture_alignment=1 (00000000000000000000000000000001)

State 15 file <built-in> line 260 thread 0
----------------------------------------------------
  __CPROVER_architecture_memory_operand_size=4 (00000000000000000000000000000100)

State 16 file <built-in> line 261 thread 0
----------------------------------------------------
  __CPROVER_architecture_single_width=32 (00000000000000000000000000100000)

State 17 file <built-in> line 262 thread 0
----------------------------------------------------
  __CPROVER_architecture_double_width=64 (00000000000000000000000001000000)

State 18 file <built-in> line 263 thread 0
----------------------------------------------------
  __CPROVER_architecture_long_double_width=64 (00000000000000000000000001000000)

State 19 file <built-in> line 264 thread 0
----------------------------------------------------
  __CPROVER_architecture_wchar_t_width=32 (00000000000000000000000000100000)

State 20 file <built-in> line 265 thread 0
----------------------------------------------------
  __CPROVER_architecture_endianness=1 (00000000000000000000000000000001)
```

```
State 21 file <built-in> line 9 thread 0
----------------------------------------------------
  __CPROVER_malloc_is_new_array=FALSE (0)

State 23  thread 0
----------------------------------------------------
  door_locked=false (0)

State 24  thread 0
----------------------------------------------------
  door_open=false (0)

State 27 file contract.cpp line 19 thread 0
----------------------------------------------------
  door_open=false (0)

Violated property:
  file contract.cpp line 21
  assertion
  door_open == TRUE

VERIFICATION FAILED
```