EPM–RT–2010-10

VERIFICATION OF SOFTWARE-HARDWARE HYBRID SYSTEMS

Neelesh Bhattacharya
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

EPM-RT-2010-10

Verification of Software-Hardware Hybrid Systems

Neelesh Bhattacharya
Département de génie informatique et génie logique
École Polytechnique de Montréal

Septembre 2010

Dépôt légal :
Bibliothèque nationale du Québec, 2010
Bibliothèque nationale du Canada, 2010

EPM-RT-2010-10
Verification of Software-Hardware Hybrid Systems
par : Neelesh Bhattacharya
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

UNIVERSITÉ DE MONTRÉAL

Verification of Software-Hardware Hybrid Systems

by

Neelesh Bhattacharya

A Research Proposal submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
École Polytechnique de Montréal

September 2010

Abstract

Verification of complex systems with multiple processors is difficult. The reason being that the generation of test cases for the whole system is quite complex. So, the system must be verified in parts and sequentially, i.e., verifying the software, hardware platform separately and the finally software running on the hardware platform. As verification of the MPSoC (Multiple-Systems-on-Chip) platform is beyond the scope of our research, we assume that the MPSoC hardware platform is already verified. Thus, we focus our research on the verification of the MPSoC software (application software running on the MPSoC platform) and the system consisting of the MPSOC software running on the MPSoC platform. Researchers have tried to verify the software portion by generating test cases using metaheutistics, constraint programming and combined metaheuristic-constraint programming approaches. But metaheuristic approaches are not capable of finding good solution as they may get blocked in local optima, whereas constraint programming approaches are not able to generate good test cases when the problem is large and complex. The combined metaheuristic-constraint programming approaches solve these limitations but lose many good test cases when they reduce the domain of the input variables. We want to generate test cases for software while overcoming the limitations mentioned. For this, we propose to combine metaheuristic and constraint programming approaches. In our approach, constraint programming solver will split the input variable domains before reducing them further to be fed into the metaheuristic solver that will generate test cases. Finally, at a later stage of our research, we want to verify the whole system consisting of an application software (DEMOSAIK or FFMPEG 4) running on an MPSoC architecture simulator, the ReSP platform. We propose to generate the test cases from the functional test objectives to check the proper functioning of the software running on the hardware platform. So, we frame the two research questions as: Verification of software by generating test cases so as to satisfy certain coverage criterion and cause the software to fail, and verification of the functional and structural coverage criteria(s) of the system as a whole. We report the results of the preliminary experiments conducted, which helps us to provide a path for the subsequent steps.

Contents

Abstract i

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Context: The domain of the problem . . . 4
- 1.3 Problem : Statement of the Problem . . . 5
- 1.4 Running Example: Software and Hardware Sample Examples . . . 6
- 1.5 Research Questions and Contributions . . . 7
  - 1.5.1 Software Verification . . . 7
  - 1.5.2 Analysis of the Interface of the Running Software and the MPSoC Platform . . . 8
  - 1.5.3 Conclusion . . . 8
- 1.6 Research Proposal Organization . . . 9

2 State of the art 10
- 2.1 Specific Literature Review . . . 11
  - 2.1.1 Metaheuristic Approaches . . . 11
  - 2.1.2 Constraint programming approaches . . . 14
  - 2.1.3 Combining metaheuristic and constraint programming approaches 15
  - 2.1.4 Verification of Hardware and Software Systems . . . 17
- 2.2 Other Related Work . . . 18
- 2.3 Discussion . . . 23

3 Proposed methodology 25
- 3.1 Test Case Generation for Software . . . 26
- 3.2 Verification of MPSoC Device with Software . . . 31

4 Preliminary Results and Ongoing Work 35
- 4.1 Preliminary Results . . . 35
- 4.2 Ongoing Work: Explanation with A Running Example . . . 36

5 Conclusion 40

6 Research Goals and Timeline **42**
6.1 Short Term Goals 42
6.2 Medium Term Goals 43
6.3 Long term goals 43
6.4 Research Timeline 43

List of Figures

1.1 A triangle classification program [1] 6

3.1 Verification of Software–Hardware Hybrid Systems 25
3.2 Software Verification 28
3.3 Verification of the Software running on the Hardware Platform 32

4.1 Input Variable Domain Splitting and Reduction Approach 37

List of Tables

4.1 Comparison of metaheuristic and constraint programming approaches on TRIANGLE code 36

6.1 Research Timeline 43

Chapter 1

Introduction

In any software project, verification and validation accounts for more than 40 % of the total development cost and effort [2]. The cost and effort required increases further when we take into account hybrid hardware–software systems, where the verification of the hardware consumes roughly 70 % of the testing effort [3]. The time-to-market and the quality of the end-product of a system are the two important parameters that needs to be improved. The challenge in the improvement of the mentioned parameters is the combination of the hardware and software verification techniques. The automation used in the present world have posed two challenges: Verifying both the components separately and proving the compatibility of the system when the software runs on the hardware platform.

Research works have addressed the issue regarding the difficulties of dealing with multi-core system applications [4]. In spite of this, no specific work has been able to come up with proper verification techniques for the hardware-software interface. Therefore, we want to take up the task of verifying the hardware-software interface so as to overcome the limitations of the established approaches. In most of the cases, after verifying the software and hardware separately, problems emerged when both were integrated together. Software–hardware hybrid systems can be tested by generating test cases from the structural and functional specifications of the system as a whole. But the approaches used by researchers working in this domain have not come up fruitful results. The above mentioned considerations motivated us to propose a research work aiming at verifying mixed hardware-software systems implemented on a multi-processor system on chip, with the intent of generating test input data that will lead to the system failure.

Another reason for considering this problem is the ever-increasing cost of multiprocessor chips. Multi-Processor Systems on Chip (MPSoC) are examples of systems that have multiple processors with single or multiple cores. Due to their nature and complexity, it

is expected that testing and verification of MPSoC software applications would amount to a substantial part of the development effort and cost. Moreover, MPSoC have a distinct hardware–software interface, *i.e.*, the interface between hardware and software. Software–hardware interface is the point of contact between software and hardware. As both software and hardware have different specifications, it is necessary to check if both are compatible with each other, i.e., if both can execute a task together, sharing resources. Some software are hardware dependent and therefore may not perform well when running on a hardware platform which does not conforms to its specifications. MPSoC platforms are also a hardware platforms which are reaching the complexity of sub-networks of communicating processors including local area network, I/O devices (e.g., static and dynamic memory, DAC and ADC), buses and/or arbiters, and software applications [4, 5], which heavily depend on the hardware-software interface. Due to the interface mismatch of hardware and software, a software running in isolation might not behave functionally correct when run on a hardware platform. Therefore, verification of such a system would lead to additional cost and effort. So, we can mention that our research work aims at the analysis and verification of MPSoC software, as well as considering the hardware–software interface-level verification, because more and more functionality are implemented by the hardware [6].

We would like to tackle some of the above mentioned problems regarding MPSoC software. To do this, we would deal with an MPSoC platform, Reflexive Simulation Platform (ReSP). ReSP is an MPSoC simulation platform working at a high abstraction level in comparison to single processor hardware platforms; components used by ReSP are based on SystemC and TLM hardware and communication description libraries. ReSP provides a non-intrusive framework to manipulate SystemC and TLM objects. The simulation platform is built using the Python programming language; its reflective capabilities augment the platform with the possibility of observing the internal structure of the SystemC component models, which enables run-time composition and dynamic management of the architecture under analysis. The full potentialities offered by the integration between Python and SystemC are exploited, during simulation, to query, examine and, possibly, modify the internal status of the hardware and software models. These capabilities simplify the debugging process for both the modelled hardware architecture and the software running on the platform.

Preliminary to checking the hardware–software compatibility issues, we must address some of the issues related to the quality assurance of software. Satisfying specific coverage criteria and generating test cases to fire unwanted conditions in the program like buffer overflow are directly related to the quality aspect of software. These challenges are posed by the software itself, in isolation with the hardware, assuming that the hardware implementation comply with specifications. The reason being that if the software

itself is not fully verified, there is no use of running and testing an incorrect software on the hardware platform. One of these challenges is the efficient generation of test input data that will cause a software to fail and the system to crash due to some unwanted conditions, as an example divide-by-zero, buffer overflow etc. The challenge of software verification is to effectively generate test input data that will fire specific kind of exceptions present in the software, e.g., division-by-zero, buffer overflow or null pointer exception.

At this point, we would like to explicitly explain the link between generation of test cases to satisfy specific coverage criteria and fire exceptions. A software may either have single or multiple guards or may not have one. A program with guards has try-catch blocks (for handling exceptions). In order to form a link between coverage criteria and exceptions, we need to transform the guard condition of the catch block to a decision node in the instrumented code that aims to generate test cases satisfying a coverage criteria [7]. While doing so, the test cases would automatically fire exceptions because the exceptions (predicate conditions in the instrumented code) are already traversed when the test cases satisfy branch coverage criterion (coverage criteria). For the programs without guard, we assume that a try-catch is either present or can be added. The open research issue, as mentioned in [8], is that the guard condition in the catch block may be very complex and so, handling it for any transformation might prove to be very difficult. As this issue is out of the scope of our research, we assume that the condition is manageable and can be transformed. There may be more kinds of exceptions in the code, but we would fire only the kinds of exception chosen before to be executed. In fact, specifying the kind of exception to be fired can also be treated as a part of the research task.

1.1 Motivation

Looking from the point of view of the people involved, we can say that every research is carried out for the benefit of a section of the society, which varies from the application domain of the technology being developed. We focus our work on providing a technological advancement in the domain of multiprocessor systems on chip for three major groups of people, as follows:

Researchers – The problem in hand is a challenging one that seems to have not been produced an efficient solution. It has gained interest of the researchers around the world for quite a long time. So, we would like to take up the challenge and come up with fruitful results.

Software and Hardware Developers – The ideas developed by the researchers would be used by the various developers to practically implement the multiprocessor system on chip concept in their respective organizations. So, they are in a sense dependent on the researchers to provide new ideas. The reason being that the developers use the ideas of the researchers to come up with products with added features.

Common User – Availability of chips at low cost has always been of interest to the common user. So, once a new technology emerges into the market, cost of the new product tends to be high. With gradual spread of knowledge about the technology, the production of these items increases manifolds, thereby reducing the cost substantially. For this reason, the user would always have an eye on the technological development.

We can easily visualize that researchers, software–hardware developers and common user are related to each other and would be largely impacted by a technological change in the domain of embedded systems.

Finally, due to the lack of success of the researchers in this domain, we wanted to take this as a challenge and come up with a novel approach of system verification consisting of small steps which will be discussed later in this and the following chapters.

1.2 Context: The domain of the problem

In some of the recent works, both metaheuristic and constraint programming approaches were combined [9, 10] together to generate test cases for software verification. But the approaches had certain limitations due to the fact that their work was based on reducing the input variable domains, which resulted in the loss of many good test cases (test cases which satisfied the coverage criterion in less execution time) when the variable domain was huge. In some works, test cases were generated by specifying certain coverage criterion. Research works in the relevant domain was mainly accomplished by the use of metahauristic approaches, constraint programming, functional verification and abstraction exploitation. So,we would explore these approaches in depth while carrying out the research work:

- Metaheuristic approaches – Metaheuristic approaches are computational methods that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality [11, 12]. These approaches make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions.

- Constraint programming – Constraint programming is a programming paradigm wherein relations between variables are stated in the form of constraints. It integrates concurrent constraint programming, constraint logic programming and functional programming [13]. Constraints differ from the common paradigms of imperative programming languages in that they do not specify a step or sequence of steps to execute, but rather the properties of a solution to be found. The difference makes constraint programming a form of declarative programming.

- Functional verification – Functional verification is the task of verifying that the logic design conforms to specification. So, it is the job of testing if the proposed design does what it is intended to do.

- Abstraction exploitation – Abstraction is a concept not associated with any specific instance object or system [14]. Exploitation of abstractions is an approach to reduce and factor out details so that few concepts can be resolved at a time.

1.3 Problem : Statement of the Problem

Our problem is the process of verifying the software running on a hardware platform, along with the verification of the interaction between both of them. The whole process is an incremental one, the succeeding stage, *i.e.*, verification of the interaction between hardware and software, depending on the outcome, i.e., proper verification of the software verification stage, of the preceding one. Thus, we have divided the major problem into the following steps:

- Test cases would be generated for a software system by the combination of meta-heuristic and constraint programming approaches, to satisfy specific coverage criterion (branch coverage or specific path coverage)and raising exceptions leading to the software failure(Section 1.2).

- Verification of the interface between the MPSoC device and the application – The software running on the hardware platform would be verified by extracting the abstractions at various levels (component, functional and architectural level). Test cases would be generated for the functional and structural verification of the system (section 1.2).

Details of the small research goals and the ways to tackle them are mentioned in Section 1.5 and Chapters 3 and 4.

1.4 Running Example: Software and Hardware Sample Examples

In this section, we present a piece of the TRIANGLE sample code [1] to use as a running example (for the software part) and ReSP platform (explained in details in the introduction section of this Chapter) as the hardware running example throughout this research proposal. This code has been used (by other researchers also) to carry out the experiments and obtain preliminary results.

```
 s    int tri_type(int a, int b, int c)
      {
          int type;

 1        if (a > b)
2-4       {   int t = a; a = b; b = t;   }

 5        if (a > c)
6-8       {   int t = a; a = c; c = t;   }

 9        if (b > c)
10-12     {   int t = b; b = c; c = t;   }

 13       if (a + b <= c)
          {
 14           type = NOT_A_TRIANGLE;
          }
          else
          {
 15           type = SCALENE;
 16           if (a == b && b == c)
              {
 17               type = EQUILATERAL;
              }
 18           else if (a == b || b == c)
              {
 19               type = ISOSCELES;
              }
          }

 e        return type;
      }
```

FIGURE 1.1: A triangle classification program [1]

The code takes three inputs a, b and c as the three sides of a triangle. Depending on the input values of the three parameters, the code categorizes a triangle into equilateral, isosceles or scalene. The code is a good running example considering the fact that it has numerous branches and reaching the innermost statement is quite difficult. Therefore we use this code to generate test cases to cover all its branches, thus reaching the innermost statements.

So, the sub-problems mentioned in section 1.3 can be explained using the running examples as:

- Verification of the TRIANGLE code by generating test cases to satisfy branch coverage and specific path coverage criterion.
- Verification of the system comprising of the TRIANGLE code running on the ReSP platform by generating test cases to satisfy coverage criterion.
- Verification of a modified version of the TRIANGLE code or another code that has some exception conditions, by generating test cases to satisfy coverage criterion or cause the software to fail in the presence of exceptions.

1.5 Research Questions and Contributions

The following sub-sections (Sections 1.5.1, 1.5.2 and 1.5.3) specify the broad outline of the research questions that we would like to focus on. We would build our schema to tackle these questions in the next chapters.

1.5.1 Software Verification

The verification should be addressed at an early stage of the software development because it affects the quality assurance aspect of the software and the system. Software verification can be done by generating test cases to satisfy specific coverage criteria(s) and raise exceptions conditions for the system to fail. Test case generation is a means for exploring the faults in software and thus verify its functionality. The generated test cases should satisfy certain coverage criterion. Research work carried out so far in the test case generation of software systems have been incomplete in the sense that either they have not shown a good performance when solving complex problems (constraint programming approaches) or they get stuck in local optima during the search (metaheuristic approaches). Further, when both metaheuristic and constraint programming approaches were combined, there was loss of good test cases because the approaches used domain reduction of input variables. Thus, we want propose an approach, building on the combined metaheuristic and constraint programming approach in a novel way (explained in section 3.1) to address the limitations mentioned by answering our first research question, which can be stated as follows:

RQ1: How can we improve the quality of software component that would run on the MPSoC platform, so as to satisfy specific coverage criteria(s)(branch coverage, specific path coverage, or specific def-use pair coverage)and generate cases by combining metaheuristic and constraint programming leading to the system failure, and overcome the limitations of the established approaches?

1.5.2 Analysis of the Interface of the Running Software and the MPSoC Platform

The task of MPSoC device verification is never complete on any multiprocessor system until we prove that the software running on the MPSoC platform is compatible with it, *i.e.*, both the software and the platform perform failure-free executions. Research works to date have not been able to come up with proper verification methods of such processors [4]. This is because the interface, or the exact point of contact between the hardware and software component in a system is not easy to handle. Chips, ADC/DACs or RAM acts as hardware–software interface in many systems. Therefore, we would like to come up with an approach of generating test cases from the functional and structural requirements of the hardware–software hybrid system. We are interested to exploit the abstractions at the various levels of the system, as well as generate test cases to verify MPSoC system. Further, we would like to verify the system by running it on a single processor (to start with), and slowly adding more processors so as to verify more complex system incrementally. The generation of test cases would in turn take care of the functional and structural criterion of the system under test.

As a practical application, we would use DEMOSAIK and MPEG 4 code to run on the MPSoC platform ReSP. A DEMOSAIK algorithm [15]is a digital image process used to reconstruct a full color image from the incomplete color samples output from an image sensor overlaid with a color filter array (CFA). To start with, we would verify the DEMOSAIK (and MPEG 4) application in isolation to the platform by generating test cases which would satisfy coverage criterion and cause the application to fail. Then, we would verify the hardware platform (ReSP) separately by explicitly defining the hard and soft constraints of the platform. Finally, we would run the DEMOSAIK application on the ReSP platform and verify the system (structural and functional verification) by generating test cases for the system as a whole. As a running example, we will start with testing the sample TRIANGLE code (Section 1.4) when it runs on the MPSoC device platform. Keeping this in mind, we present the third research question as:

RQ2: How can we check the compatibility of software running on the MPSoC platform, thereby verifying the functional and structural coverage criteria(s) of the system as a whole, as well as ensuring that the system meets the imposed constraints (e.g. timing and bandwidth constraints)?

1.5.3 Conclusion

On the accomplishment of the research questions, we would like to have the following contributions of our research:

- Efficient generation of test cases (satisfying test coverage criterion) for the software under test by the combination of metaheuristic and constraint programming approaches to satisfy specific coverage criterion and raising exceptions leading to the failure of the software application to be run on the hardware platform.
- Satisfying the structural and functional criteria(s) of the system as a whole (comprising of the software running on the hardware platform), while exploiting the various levels of abstractions of the system.
- Successful verification of the overall system by applying the approach on the DEMOSAIK and MPEG 4 codes running on the MPSoC platform.

1.6 Research Proposal Organization

This research proposal is divided into six chapters. Chapters 2 to 7 have been organized in the following manner, Chapter 1 being the introduction:

Chapter 2 briefs the state-of-the-art, which throws light on the various relevant literatures in the fields of constraint programming, metaheuristics, combined approaches of metaheuristic and constrained programming, verification of hardware and software systems along with their interaction.

Chapter 3 describes the proposed methodology for tackling the two research questions mentioned in Chapter 1. The approaches for each research question have been divided into steps to explain the methodology in detail.

Chapter 4 reports the preliminary results of the experiments conducted so far while trying to address Research Question 1, along with providing a more in-depth analysis of the proposed approach for answering this question.

Chapter 5 concludes the overall proposal by stressing the aspects of the proposed approach and explicitly mentioning the expected outcomes of the research work.

Chapter 6 points out the target milestones set to answer all the research questions. Classifying into short term, medium term, and long term goals, this chapter provides a flow of the research in terms of the set goals. Further, the chapter also briefs timeframes (tentative start and finish time) for achieving the various goals mentioned , including the duration to accomplish each of these tasks.

Chapter 2

State of the art

As discussed in Chapter 1, we want to tackle the two major problems: the generation of test cases to verify software and the interaction between the two when the software is running on the platform. To propose novel approaches to tackle the two problems, we survey the work carried out by researchers in the the domains of generation of test cases by metaheuristics and constraint programming approaches, and the verification of mixed software–hardware systems.

In most approaches developed so far, test case generation for software has been primarily been relying on metaheuristic and constraint programming. Some researchers shifted their focus on combining both of them to obtain the best of the two worlds, i.e., overcome the limitations of constraint programming (scalability issue for complex problems) and metaheuristics (getting confined to a small part of the search space). Though the works did overcome the limitations of the two approaches, they failed to generate good test cases, due to the loss of many good solutions while reducing the input domain variables to a large extent. To verify software-hardware hybrid systems, researchers also have come up with efficient approaches like stimuli generators, symbolic execution, concolic testing, etc. Therefore we focus our literature survey in the following areas:

- Metaheuristic approaches applied in various applications, specifically for software test case generation.
- Constraint programming applied in various applications, specifically for software test case generation.
- Combining metaheuristic and constraint programming approaches for test case generation of software.
- Verification of software and hardware systems.

The following two section summarizes research works carried out by researchers in the above mentioned areas. The literature review has been divided into two sections. The first section consists of the literatures that either motivated us to take up the problem or had specific limitations that we aim to overcome by our proposed approach. The second section details some of the other research works carried out in the relevant domains, though we are not interested to build on or overcome any of the limitations of these works. This section provides us with the knowledge of how the various approaches have been used in various applications.

2.1 Specific Literature Review

We now explore the literatures that are relevant to our problem. We would either like to build on their work, or would use some of the important concepts mentioned to fulfil our research goals.

2.1.1 Metaheuristic Approaches

Harman *et al.* [16] provides a detailed overview of evolutionary testing methods for embedded systems. The technical features and special requirements of embedded systems make them complex and testing them becomes more difficult to automate due to several reasons. This complexity gives rise to the need of evolutionary testing that are metaheuristic search methods used for test case generation. The goal of the metaheuristic search methods is to successfully generate new generation of test cases with better combinations of the particular parameters (like fitness function values, execution times, etc.) that affect the overall quality of the software. The concept of fitness function is explicitly used in this context. Applying search method software testing, the test is transformed into an optimization task, to generate test cases. The optimization is followed by changing the initial population of test cases, evaluating the fitness, selecting the individuals, recombining them and, finally, mutating them to obtain a new population. Structural testing methods like node-oriented, path-oriented, node-node oriented and node-path oriented methods can be automated by splitting up the test into partial aims. Experimenting with several sample codes and setting the branch coverage as the coverage criteria, the authors [16] successfully meets the aim as 100% on numerous occassions. Safety testing is another kind of evolutionary testing. The aim of safety testing of Evolutionary Testing is to generate the input conditions that would lead to the violation of the safety requirements. The fitness function in this case is based on the pre and post conditions of components of the system, like speed limit, petrol consumption, etc. Experiments were conducted on six engine control system tasks and, as expected,

the systematic testing took a longer execution time (execution time is a criteria in our problem) compared to evolutionary testing (ET) in all cases. Thus, evolutionary testing proves to be better than systematic.

To analyse the problem of investigating the structure of a program that enables evolutionary testing to perform better (less execution time) than other approaches like constraint programming and random algorithms; and the crossover form that can satisfy the coverage criterion for such program structures, McMinn [17] exploits the concept of constraint-schema. Four well-defined constraint-schema types (Covering, Fitness-Affecting, Building Block, and Contending) have been analysed and some of its salient features have been summarized as follows: For the crossover to have a larger impact on the search progress, there should be a significantly large number of conditions in the predicates, i.e., more number of input predicate conditions. More input predicate conditions create a possibility of reaching a more specific constraint by doing crossover of a set of input conditions. It is important to have conjuncts with disjoint set of variables so that the crossover has an impact on the search. In other words, the conjuncts should reference to different input variables. Further, if they are nested, it is impossible to execute the input conjuncts in parallel. Specifically McMinn stressed on the facts that it is not easy to precise the exact number of conjuncts required for the crossover to have an impact and conjuncts with non-disjoint sets of variables definitely reduce the effectiveness of the crossover.

Harman *et al.* [18] finds a correlation between the input search space and the performance of a search based algorithm. They explored the impact of the domain reduction on random testing, hill climbing and evolutionary testing approaches. Starting with a brief introduction of random testing, hill climbing and evolutionary search-based strategies, they emphasize on the input domain reduction by stating that it is carried out by the removal of irrelevant input variables from the search for each branching node. A variable dependence tool, VADA is used for this purpose. They proved mathematically that input domain reduction would not affect random testing as much it would do to hill climbing and evolutionary testing. To answer the already posed research questions, they conducted several experiments to find the impact of the reduction on each strategy in isolation. As expected, there was no significant change found in random testing strategy using the reduction. In contrary, both hill climbing and evolutionary testing found a reduction in effort to find a test data after the domain reduction; though there was not much significant change in the effectiveness. While comparing the performance of hill climbing and evolutionary testing, it was found that evolutionary testing benefits more from the reduced domain than hill climbing. Improvements in terms of the actual number of evaluations is lower in hill climbing than evolutionary testing.

Tracey *et al.* [7] details a dynamic global optimization technique to generate test cases for raising exceptions in safety critical systems. Random, static and dynamic test data generators are the three main classes of automatic test data generation. But, as exceptions occur rarely, this aspect has not been taken care off concretely by these approaches. The author stresses that proving a software is free from exceptions is easier than checking the exception handlers in the code. Predefined exceptions like constraint error, program error and storage error are necessary to be handled. Optimization techniques like genetic algorithms, once employed by the test case generator, be guided to locate the test data. This is proposed to be done by the fitness function, which depicts how close an input data is to raise a certain exception. Once found, manipulations are done to guide the search along a specific test path. The search technique would help to raise the exception condition, while the branch predicate functions would guide the search so that the test is executed along a desired path through exception handlers. Though the SPARK-Ada tool helps to check if a sample code is free from exceptions, it is not efficient to do so for all kinds of systems. For safety critical systems, the author proposes to integrate the test data generator into a process that shows the software to be free from exceptions. This would reduce cost and complexity by not executing operations like overflow, underflow and division-by-zero. The approach was tested in two parts: by collecting small Ada95 programs to generate test data to raise particular exceptions and specific exception condition coverage. In addition, the check for being free from exceptions was done using the code for a critical aircraft engine controller. The results were promising and the approach could generate test test cases that caused exceptions and covered the exception handling code efficiently. The author suggests that there are potential open research questions regarding the impact of the various tunable parameters of the optimization techniques on the quality of the test data in a complex search space.

The research works mentioned are important to our research. The work by Harman *et al.* [18] pointed out the important limitation of the metaheuristic approaches getting blocked in the local optimum. Apart from this limitation, he has also depicted the usefulness of the use of metaheuristic approaches in generating test cases for embedded systems. The concepts of genetic algorithms and hill climbing have been presented (along with comparing its performance with other approaches), thus motivating us to incorporate metaheuristic approaches in our research so as to solve the limitation of getting stick in the local optimum. Tracey [7] pointed out the open research issue regarding the complexity of the guard condition in a program, while dealing with exception conditions and thus motivated us to develop a link between satisfying specific coverage criteria and test cases generation to fire exceptions. McMinn [17] provides us with the knowledge of the impact of predicate constraints on the program structure. It is useful for our research because to verify a software by generating test cases, we should have the knowledge of

the impact of the predicate conditions on the program structure when a generated test case, to satisfy coverage criterion like branch coverage, encounters certain predicate constraints. The knowledge would help us in comparing various test cases. As our research deals with software verification, we will deal with variables with huge domain size which might be difficult to handle. So, it would be beneficial if we use a tool which can reduce the domain of the variables. Harman *et al.* [16] provides us such a tool, named VADA. Further, the work provides the knowledge of the effect of reducing input domains to the performance of various algorithms. So, in our research we would reduce the relevant sub-domains using the VADA tool. At this point we would like to emphasize that the use of VADA in our research is just one of the many steps we propose to carry out while addressing the first research question. This is because our approach would lead to the task of splitting the individual input domains, checking the constraint satisfaction each sub-domain w.r.t. the predicate conditions and removing the irrelevant input sub-domains before eventually reducing their domains using VADA. Our approach differs from the established approaches in that we do not propose to reduce the input domains directly. We propose to carry out more computations (mentioned in section 3.1) for the generation of test cases which satisfies the coverage criteria while firing some exceptions. Therefore the use of VADA tool is just a small part of the numerous steps we wish to execute .

2.1.2 Constraint programming approaches

Refalo [19] presents a new approach of using the impact factor in constraint programming to overcome the limitations posed by integer programming. The author clearly points out the demerits of integer programming and backtrack search strategy. He explains impact as the measurement of the importance of an assignment statement on the search space reduction. If the overall search domain is considered as the product of the individual domain size, the reduction rate due to the impact of the assignment is calculated by the simple ratio of the overall search domain after and before the assignment. Similarly, the impact of a variable is considered as the average of the impacts of the individual variables. To initialize the impacts, the domain of a variable is divided into sub-domains and the individual impacts has to be calculated. The process needs to restarted so as to provide equal opportunities to the whole search space to be explored at the beginning of the search. The experiments conducted on the multiknap sack, magic sequence and latin square completion problems showed a significant reduction in the domain size and required minimal overhead. Thus, the work motivates us to use input variable domain reduction as it has a positive impact on the generation of test cases and the overhead is minimal.

Levis Paolo [20] also addresses the exam scheduling problem, but from a constraint programming approach. He proposed a three phase approach (three different variables involved), i.e., time slot variables, room variables and invigilator variables to address the problem on hand. A set of seventeen hard constraints were chosen initially. To have a better performance compared to a manual approach, the author introduced a constraint relaxation system that further involves a constraint hierarchy of the labelled constraints. For the hard constraints, an evaluation formula is generated. Similarly, soft constraints were used to generate another evaluation formula involving the individual weights and a function value for each soft constraint. Both of these were summed up to have the global evaluation function for further minimization. Four labelling strategies were used based on the combination of the three variables mentioned before. The standard branch-and-bound algorithm was used to deal with the optimization problem and dynamic ordering method for variable ordering. Data from the University of Fernando Pessao was used to conduct experiments. The approach gave a solution three times better than the manual approach. Particularly the soft constraints created the difference whereas two strategies (R− >T− >I) and (T− >R− >I) could not provide good results.

The research work suggests that constraint programming approaches have had success in generating test cases for software, but with the limitation of failure to generate good test cases when the problem is huge (large number of lines of code) and complex (too many and complex predicate conditions). Refalo [19] encouraged the use of domain reduction by mentioning that their work achieved significant reduction with minimal overhead. He also points out the limitation of constraint programming to fail when the problem is huge (in terms of LOCs) and complex (in terms of the constraints). Thus, we would propose an approach that will overcome this limitation. Levis [20] motivated us to use constraint programming in practical applications (exam scheduling) and provided with the knowledge of specifying hard and soft constraints for such applications. We would use use this knowledge to specify hard and soft constraints for the verification of the hardware platform in our research work.

2.1.3 Combining metaheuristic and constraint programming approaches

Quite a number of research papers have been published on the combination of metaheuristics and constraint programming approaches like [9, 21**?**]. In this section we focus on some of the specific work that are relevant to our research questions and helps us to answer the questions.

Castro [22] proposed a hybrid approach consisting of constraint programming and metaheuristic to overcome the limitations of both the approaches. The proposed approach is

based on the judicious use of the communication between the incomplete solver (i-solver) and the complete solver (c-solver) depending on the direction of the communication, *i.e.* both the constraint programming and metaheuristics solvers are executed one after the other (after the order of the use of the solvers is decided) to communicate with each other. The iterative approach is initiated by solving the CSP for the optimization task using the c-solver. This solver would reduce some of the branches using the bound for the objective function, thus reducing the search space. This reduced search space, with a bound for the optimal value of the problem is provided to the i-solver to search a local optimal value in the given domain. The approach executes in the reverse order if the direction of communication is c-solver to i-solver. The whole process continues until the problem becomes unsatisfiable, specific time limit is reached or no solution is found. Experiments were conducted on the vehicle routing problem and both the cooperation schemes provided the same best result, which was much better than using constraint programming in isolation. When using it a few times before hill climbing, the scheme always took less time than the approach involving the other direction.

Tuwasi-Anh *et al.* [23] addresses the problem of developing examination timetables for educational institutions by using a two stage approach. They propose a hybrid approach involving constraint programming and simulated annealing. The constraint programming approach provides an initial solution, whereas simulated annealing would improve the existing solution. Thus, both hard and soft constraints would have been taken care off. After choosing the specific constraints, they were divided into hard and soft constraints. Constraints like exam clashing, room clashing were categorized as hard constraints, whereas student restrictions, room restrictions, release restrictions, room utilization and predefined room constraints were partitioned as soft constraints. The backtracking with forward checking algorithm was used in the first stage and a priority score was assigned for each exam. While applying the simulated annealing algorithm, a variant of the kempe chain neighbourhood was used. A penalty score, based on the distance between two exams was assigned as the cost function whereas a geometric cooling was used to take care of the cooling schedule. Real data from HCMC University of Technology was used to run experiments and the constraint programming stage took less than 2 minutes run, providing a good feasible initial solution. Simulated annealing phase run times varied between 326 and 410 seconds. Anh concluded that the longer the algorithm ran, the more improved the solutions were.

Both the research works mentioned above suggests that two-solver approach (metaheuristics and constraint programming solvers) have been used to combine the approaches by the reduction of input variable domains. Though the approaches solve the individual limitations of metaheuristic and constraint programming approaches, when the domain of the input variables is very large (in the range of 232), there is a possibility that the

reduction of the domain would result in the loss of good solutions or test cases. Further, the experiments conducted on programs that were small in size and had small number of constraints. So, the approach would falter when generating test cases with the big application software. We would like to overcome these limitations while carrying out our research.

2.1.4 Verification of Hardware and Software Systems

An interesting article by the Electronic Engineering times [24] talks about the present and future network on chip topologies. Network-on-chip (NoCs) suffers from the limitation of maintaining a high bandwidth required for processors, memories and intellectual property (IPs) blocks. But it excels in the features like replacing fixed bus with pack-based approach and layered methodology; along with its ability to predict latency and arrival of packets at the chip level. Therefore, it has faster data transmission rate, more flexibility and easier intellectual property reuse. Among the issues that needs to be sorted are the granularity of the cores, bus width, network topology, etc. Overall the researchers suggest that NoC may be difficult in specific applications due to its big die size and power consumption factor. A new technology, STNoC supports any topology and could be generated automatically with any tool. Sonic MX is another recent technology claiming power efficiency as its major factor. MX offers packetization and adds more power management features. One of its recent competitors, Arteris is a faster approach, more configurable and independent of a specific protocol. It has a transaction layer, a package transport layer and a physical layer. It claims to provide a 3-4 times better efficiency than any present bus. But being a synchronous system using long wires, a fully asynchronous system would be difficult to implement. Arteris and Sonics are the two present big competitors, Arteris focussing on networking and Sonics on mobile handsets.

The book [5] dedicated to the design of embedded systems presents various aspects of MPSoC modelling and data mapping tools. Clarifying the reasons for the rise of the development costs of Systems-on-chip and the need for the domain-specific flexible platform, the authors specified the limitations as well as advantages of the three established programming models, namely, SMP, Client-server, and Streaming model. Emphasizing on the assignment and scheduling of the streaming applications by the MultiFlex mapping tools, the authors proposed a stepwise mapping refinement tool. The tools minimize the processing load variance and the communications between the network-on-chips. A 3G base-station was chosen as the application. Another important aspect mentioned is the need for the retargetable system in terms of task partitioning architecture change. The established platforms like MPI or Open MP are not exactly retargetable in the

correct sense. So, they propose the CIC retargetable programming model that specifies the design constraints and task codes. The mapping algorithm considers temporal parallelism along with functional and data parallelism. An H.263 decoder was used for the experiments and the design productivity was improved. Future research directions like optimal mapping of CIC and the exploration of the target architecture have been mentioned clearly. Later, they defined the various programming models to abstract hardware-software interfaces for heterogenous MPSoCs at the various abstraction levels like system architecture, virtual architecture, transaction accurate architecture and virtual prototype. Each of the levels have specific task and resource controls and communication primitives. They proposed a novel approach to combine the Simulink environment for high level and System C design language for low level programming. To achieve the combination, they followed a four step approach: Partitioning-Mapping, Communication mapping on hardware, Software adaptation to hardware and Software adaptation to specific CPUs and memory.

The article by EE times [24] is highly motivating for our research because it mentions the present and future hardships that the developers would be facing trying to develop multicore processors. They explicitly focus on the various issues that has led to the lack of success in the multiprocessor domain. Some of the issues like data transmission rate suffers when the interface between the hardware and software is not exlpicit. For example, if the data transfer rate of the hardware is much more than the software, the software can cope up with the pace at which the hardware executes its functions. At the same time,the hardware will not receive input from the software quickly enough to produce output in spit of it having very good processing speed. We would like to address some of these issues and provide an approach to verify such systems. To do this, we need the knowledge of modelling such systems. The book [5] details all the information required to model multiple-processor-on-chips (MPSoC) systems. It provides the various architectural models and mapping tools which we would like to use while verifying the software–hardware interaction (Research Question 2).

2.2 Other Related Work

This section briefs some of the other research work carried out in the domain of metaheuristics, constraint programming, combined metaheuristic and constraint programming, and verification of hardware–software systems. Though this work is not directly relevant to our research, they provide us with the knowledge of ways to apply the approaches in various application domains. We would use some of the concepts while addressing our research issues.

Alba *et al.* [25] focusses on the decentralized and centralized evolutionary algorithms (EAs) for the automated test data generation problem. A proposed tool, called the test data generator, creates several partial objectives, i.e., coverage criterion(s) for the immediate test execution from the overall global objectives (coverage criterion(s) for the overall test case generation) and treats each of them as an optimization problem. These problems would be solved using EAs. To get more information about the distance function, program instrumentation was carried out automatically by parsing the C source code and generating the instrumented code. The test data generation process starts with the selection of random inputs to reach some conditions. A partial objective is then selected depending on the predicate constraints (combination of all the conditions traversed by the previous input). The distance function is applied based on the partial objective (coverage criterion). The optimization algorithm then guides the search by using the distance value. The algorithm stops when a partial objective is covered and restarts to continue with a different one. Experiments were conducted to compare the performance of the following parameters: parallel decentralized EAs vs. panmictic ES and distributed GA vs. panmictic GA. An unexpected trend was observed with respect to the coverage metric criteria in that both algorithms (distributed and panmictic) had no statistical difference, whereas w.r.t. effort, the number of evaluation for distributed ES was more than panmictic ES for the same coverage. In terms of execution time, distributed ES was faster than panmictic ES, as expected. A similar trend was observed for the genetic algorithm. Another set of experiments compared the performance of ES (Evolutionary Strategy) [26] and GA (Genetic Algorithm) [27] and it was found that ES performed better than GA either with distributed or panmictic. These experiments also revealed the following facts: the version that searches different partial objectives always underperformed w.r.t. those which searches the same partial objectives; full coverage of the searched partial objective should be used as the stop criterion; single seed is the best solution for the initial population; distributed approach is not a good approach as it does not provide a high migration gap.

Zhao [28] proposes a new approach to deal with generation of test cases involving character string predicates. His approach proves capable of dynamically generating input test data using gradient descent for function minimization. Gradient descent technique is a means of minimizing the branch predicate function by adjusting the input test value to reach a minimum value for the resulting function. Starting with a random input and executing this value on a certain branch, the result is checked and a new input value is introduced by a small increment or decrement on the previous value in order to have a better adjustment. When the direction is found suitable, a larger increment or decrement is exercised with a new input and this process continues until the function becomes negative or no further improvement is possible in the function value (decrement). The

MAX program was chosen as the sample code and experiments were conducted. The code, though small, had many structures, IF-THEN-ELSE statements which made it quite difficult to analyse. The coverage of the generated test data was measured using ATAC (Automated test analyser for C) tool and it was found that 88% of the p-uses (predicate uses) were covered. While comparing the evaluation of the number of branch function in gradient descent, gradual descent and random number of test generation, gradient descent approach proved to be the most economical. Surprisingly, random number approach easily outperformed gradual descent approach as well.

The test data generation problem has been tackled with either static or dynamic approaches. But both the approaches have notable limitations. Williams [29] proposes an approach that takes the best of both the worlds. Based on the dynamic analysis, it uses a constraint logic programming for the path predicates to fin the test cases. It overcomes the complexity of static analysis and heuristic function minimization problem. The approach starts with the instrumentation of the source code, which is executed by a random test case (instrumentation step). While executing the program, each assignment is used to update and store the current values and replace these values with their corresponding symbolic values (substitution). Once done, the path predicates and conditions are recovered for the executed input data. This predicate is negated, so as to use a different input data that will execute this predicate and thus reach an unexplored path. After negation, if all conjunctions are infeasible, the longest unsolved infeasible conjunction is chosen and the process continues (test selection and constraint solving). Because loops can create combinational explosion in the number of execution paths in specific cases, a limit has been fixed to limit the number of times a loop would be executed. To take care of this criterion, the condition is set that if the negation of a condition results in a loop to have more than a fixed number of iterations, it would not be explored. The approach was validated with examples of mergesort, TRIANGLE, and Bsort. For the mergesort example, in each run 337 test were generated with running time between 0.75 and 0.81 sec. All k-paths were tested under 1 sec. For k having a value 10, 20,993 tests were generated and 15357 infeasible paths were eliminated in 116 sec.

Functional verification deals with the limitation of generating input test vector good enough to cover a large set of behaviours using limited resources. Ferrandi [30] comes up with a solution that uses the concept of transitions and generates all test vectors to exercise all possible sequences. The finite state model is used as the system under test. It is composed of several single FSMs that interact and exchange data between themselves (other FSMs). During the acquisition of data, all the topological information about the system is obtained by the model. The acquisition consists of port list (list of all input port of the system), statement list (list of data of all statement in the code), conditional instruction list (list of the conditional instructions in the code) and transitions list (list

of all possible transitions allowed in the system). In the sequence enumeration phase, all the possible sequences are listed satisfying the following criterias: breadth first (all sequences composed of a predefined number of transitions), breadth first plus transition selection (breadth first restricted to a set of predefined set of transitions) and transition coverage (covering all transitions). The analysis of sequence phase aims at generating a set of constraints corresponding to the execution of each and every sequence. An expansion algorithm is proposed, which works on expanding the the leaf of a tree once a variable on the left side of an equation has a time mark greater then the time frame when the statement is executed. The constraint equations for each sequence are translated according to the GProlog format for bit, bit vectors and integers. Finally, two cases arise: the set of constraint equations for a sequence can or cannot be satisfied. GProlog solver obtains solutions and stores them to map over the input ports along the time frames. When extended to multiple processes, the simulation time is addressed and only after all the processes are completed, signal values are updated. The approach has been tested on a set of examples written in system C. The results clearly outperformed the classical ATPG in terms of path coverage due to the use of GProlog solver.

Test case generation verification has limitations. Roy [31] provides a tool, X-Gen to solve the limitations. It accepts as an input, the types of components, their interconnections and interactions and set of user-defined requests. The system model consists of component types, configurations, interactions, and testing knowledge. Component types are the various types of hardware components used in the system and consists of the ports (connection with other components), internal state (set of resources residing in a component) and behaviour (constraints describing relationships between properties). The ways in which the components interact with each other is detailed by interactions. Interactions consist of the various actors and their relationships with other components under specific constraints. An expert knowledge is incorporated into the system model to find out the fault-prone areas. The heart of the model is the file that acts as the template (request file) for the testing, characterizing and describing the verification goals. The order of executing the interactions is controlled by sequence, mutual exclusion or rendezvous constructs. The generation process consists of traversing the high level statement tree and generating the interactions at the leaves. To refine the overall process, the XGen generates an abstract test and then executes it through a single refinement stage. Being a tool which provides an additional level of abstraction for the system and extensive aids for modelling testing language, it finds wide application with engineers and users writing configuration and request files.

Backer *et al.* [32] solves the Vehicle Routing Problem, by combining Constraint Programming and Metaheuristic Approaches. Vehicle routing problem is the construction

of a set of routes for the vehicles to minimize the cost of operations, given a set of customers requiring a visit, and a fleet of vehicles that can perform the visits. The author did not use the conventional depth-first search and branch and bound for the constraint programming approach. Instead, he used constraint programming for checking the validity of the solutions. Depth-first search is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children [33], whereas branch-and-bound is a like the breadth first search [33] for the optimal solution, but not all nodes get expanded (*i.e.*, their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.Two specific representations used for the implementation are the passive representation (template or a database which holds in the current solution) and the active representation (template that holds the constraint variables where the propagation takes place). The objective function was chosen to be the cost of the vehicles and the distance travelled by them; whereas the constraints imposed were divided into capacity (weight, volume) and time constraints (when the customer will arrange a visit, length of the routes). The local search procedure was succeeded by metaheuristic approaches like tabu search and guided local search. Five specific move operators were used for the local search. The experimental results were quite satisfactory. Fast guided local search and guided tabu search performed much better than single tabu search, as expected. When compared with the other approaches, guided local search had the worst performance in comparison to guided tabu search, which performed much better.

Koushik Sen [34], in his paper, introduces a new concept to automate the generation of test input data by concretely and symbolically executing a program simultaneously. The main idea is to generate all the feasible paths in a given program that has memory graphs as inputs. Memory maps are logical input maps that represents all the inputs in a program. A hybrid concept of concrete and symbolic testing, concolic testing, effectively removes the limitations of random and symbolic execution testing. It is implemented by instrumenting a program and inserting suitable function calls where required. The function calls serve the purpose of executing the program symbolically, thus nullifying the need of using a symbolic interpreter. Relying on the depth-first strategy, the algorithm executes the program concolically, before negating an encountered path constraint so as to generate a new set of test input data that would lead the execution along a different path. The process continues until all the feasible paths have been explored.

Sen [35] has extended his brief overview of concolic testing, as mentioned in the research work of Kaushik [], in this paper. He explores concolic testing in a much detailed manner, after imposing on the clear limitations of the random and symbolic testing

approaches. Further, he proposes three concolic testing strategies and compared their execution and performance with the traditional depth-first search strategy. Initiating his in-depth analysis with the bounded depth-first search approach, he explaines that once a partial aim (a test path in this case) has been chosen, the execution would negate the first encountered constraint and generate a set of test cases. This would in turn lead the execution to a different path and the process would continue until all the feasible paths have been explored. The control flow directed search employs the concept of minimum distance of every branch from one of the targets. Based on the execution of the program, the approach would compel the execution along the branches with smaller distances, thus forcing the execution along a different encountered path or branch. Uniform random search uses random paths instead of random inputs. While traversing the random path, the strategy would decide the direction of movement on a probabilistic basis; thus reaching new paths. Finally, in random branch search, a branch along the current path is chosen at random and the execution is forced to take another branch. Experiments were conducted on sample programs like Replace, GNU Grep 2.2 and Vim 5.7 whose LOCs varied from 600-1500. It was found that CFG directed search and random branch search performed much better than random testing and depth-first concolic search.

2.3 Discussion

These works provide some of the interesting and salient features that we would like to exploit to answer the research questions mentioned in Chapter 1:

- Metaheuristic approaches have been largely proved to be incomplete, as they have the tendency to get stuck in the local optima. Therefore, generating test cases based on these approaches has been unable to provide good solution.
- Constraint programming approaches have proved to be able to find good solutions, i.e., the generated test cases have satisfied the given criterion like branch coverage, selected path coverage etc. But the problem with constraint programming is scalability, i.e., it fails to generate genuine test cases for large and complex problems. Thus, in spite of it being considered as a complete approach, its limitations (scalability problem) overcome its benefits.
- Most of the work related to the combination of metaheuristic and constraint programming approaches have used both the constraint programming and metaheuristic solvers by reducing the domain of the input variables after removing the irrelevant input variables from the effective search. The problem with the combined

approach arises when one deals with a large input domain. In this case, reducing the domains might cause a loss of good search space for all the input variables, and thus might lead to loss of good test cases.

- Formal methods and simulation-based techniques are the approaches used for verifying processors. The simulation-based approach has proved to be the stronger of the two (in terms of computational efficiency and execution times) and thus, has been used extensively in spite of its limitations like proper generation of test stimuli and programs.

- Though some of the problems related to the symbolic execution have been addressed properly, the problem of handling abstract data types still remains an open research question. So, symbolic execution is not a good approach to generate test cases for software.

- Use of stimuli generator is a way to address the hard and soft constraints of the hardware platform. The user request features would take care of the hard constraints, while the expert knowledge would take care of the soft constraints.

- At the various levels (component, functional and architectural level) of the software-hardware hybrid systems, extraction of abstractions are essential for its verification, because dividing the whole system into components and functions to find the abstractions eases the verification process.

We would propose our model on the basis of the points mentioned above, trying to negate their limitations and improve on the performance. The proposed model will be described in details in Chapter 3 and the preliminary results have been reported on Chapter 4.

Chapter 3

Proposed methodology

In this chapter, we want to address the steps to be taken so that we can answer the research questions mentioned in Section 1.4 of Chapter 1. Figure 3.1 depicts the steps adopted in the proposed methodology and mentions the steps that answer each research question. Each block in the figure leads to the solution of an important step of the proposed methodology and acts as inputs to the subsequent blocks.

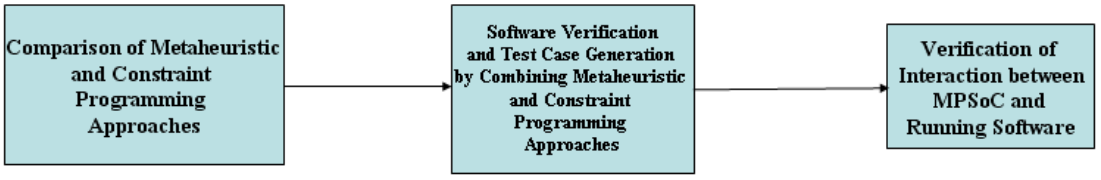

FIGURE 3.1: Verification of Software–Hardware Hybrid Systems

Before we verify the software, we want to compare the performance of test case generation by constraint programming and metaheuristic approaches. The comparison from the test carried out will be a baseline for our research. Research work in the relevant domain have generated test cases by combining metaheuristic and constraint programming approaches [23]. The approach used metaheuristic and constraint programming solvers, one following the other. But the approach had some limitations mentioned in section 2.5. So, we want to build on the approach of Duong [23] to overcome the limitations and justify a reason for following a particular order of execution of the two solvers, metaheuristic and constraint programming. We want to generate test cases by combining both the approaches and so we need to know the direction of propagation of execution for the two solvers, *i.e.*, which of the two solvers (metaheuristic and constraint programming) would follow the second one. The comparison gives an idea of this direction and is depicted by the first block. The experiment for the comparison will be carried out by executing the running example (section 1.5) with both metaheuristic and constraint

approaches separately. The execution time for each approach would decide the order of the two solvers. The solver which takes more time to generate test cases would follow the other solver. The results of the experiment cannot be generalized because the comparison has been carried out with one sample code, but definitely can provide an idea about the performance of both metaheuristic and constraint programming approaches. We will conduct experiments with more sample programs so as to generalize the choice of the order of using the two solvers. Section 3.1 provides a detailed explanation about the experiment. The second block of the figure shows the software verification part, in isolation to the hardware part, which we assume to be correct. Following the software verification is verification on the ReSP platform of the software on a simulated MPSoC device. This is to say that the final stage is to verify the software running on the platform, *i.e.*, verification of the interaction between the MPSoC device and the software application to generate test cases. The software–hardware interaction depends on the whether the software is platform-dependant or not [5], programming language used to code the software, and if the software can make use of the hardware [6]. The following two subsections explain the steps to be followed to answer the two research questions, respectively.

3.1 Test Case Generation for Software

To verify the software under test (Research Question 1), we need to generate test cases that satisfy specific coverage criteria(s) and also possibly cause the system to fail by firing exceptions. These two will satisfy the quality assurance aspect of software. Firing of exceptions is equally important because once we choose specific exception to be fired by the generated test cases, while trying to fire the exceptions with the set of test case, some coverage criterion like branch coverage and specific path coverage would be satisfied. This can be achieved by transforming the guard condition of the catch block to a decision node in the instrumented code that aims to generate test cases satisfying a coverage criteria [7]. The test cases would automatically fire exceptions because the exceptions (predicate conditions in the instrumented code) are already traversed when the test cases satisfy branch coverage criterion (coverage criteria).

To overcome the limitations of generating test cases by metaheuristics and constraint programming separately, the need to effectively combine both the approaches is quite essential. Substantial amount of research has been carried out in the domain of combining metaheuristics and constraint programming approaches [9, 10, 21]. Most of the researchers exploited the fact that constraint programming provides the initial solution

and thus solve the hard constraints; whereas metaheuristic approaches improve the solution, and thus solves the soft constraints. The regular constraints are called hard constraints, while the cost functions are called soft constraints [23]. Hard constraints are the variable constraints that should be satisfied, while soft constraints only express a preference of some solutions (those having a high or low cost) over other ones (those having lower/higher cost). Therefore, soft constraints are used mainly to improve the solution of a problem.

While combining the two approaches, the researchers had the knowledge of which of the two solvers would be executed first. Depending on a order of the solvers chosen, *i.e.*, which of the two solvers would start the process and which one would follow, the execution would either start with constraint programming to reduce the input domain (thus searching for the global optimum of this domain), and provide this reduced domain to the metaheuristic approach for finding the local optimum; or vice versa [22]. As expected, the performance, depending of the direction of the two solvers, varies. Thus, two solvers, one for the constraint programming and the other for metaheuristic approach were used sequentially for searching the whole domain. The process iterates until a certain fixed criteria, e.g. traversing a chosen branch, has been reached or the whole search space has been traversed. The domain reduction (reduction of individual domains for all the input variables) was carried out by getting rid of the irrelevant variables judiciously by the variable dependence tool VADA [36]. The VADA system computes variable dependence for the C programming language. It adopts the approach of transforming the full C language to a more manageable core language, for which variable dependence is computed.

The drawback of this approach is that it relies on the reduction of the whole domain of the individual variables. When the domain of a variable is huge, the reduction would result in the loss of numerous valuable test cases. Thus, the generated test cases will not be good enough to satisfy the coverage criteria quickly. We want to overcome this limitation by building on this approach.

Castro conducted experiments [22] using sample codes that were small in size and had large input domains. Reduction of large input domains, by removing the irrelevant input variables, can lead to a resulting domain that has already excluded some good areas in the search space, where a solution could have been found. Further, the problem increases when sample code gets bigger in size with more conditions; in this case handling large domains becomes more difficult. In large codes, the number of input variables and constraints on each of the variables are complex in nature. As a result reduction of input variable domains in such problems might lead to the removal of some of the useful search space values. Keeping these limitations in mind, we propose a novel approach of

splitting the huge input domain of the individual variables into parts, before reducing their size to be given to the metaheuristic search engine for providing better solutions. Figure 3.2 shows the flowchart of the proposed approach.

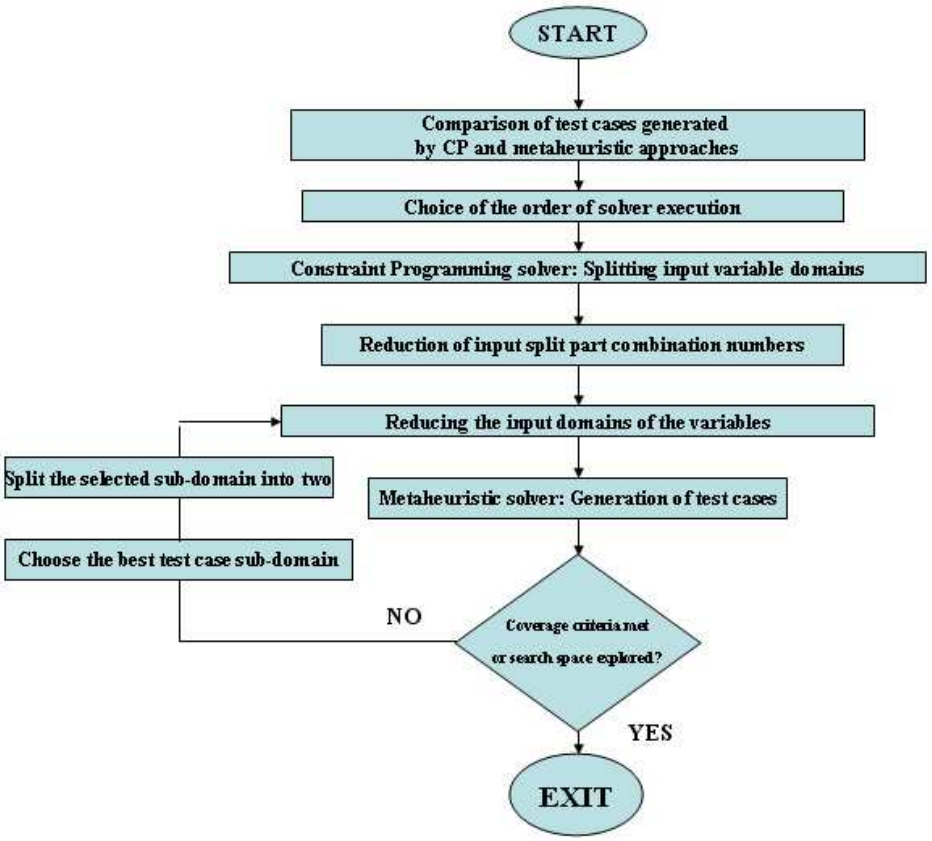

FIGURE 3.2: Software Verification

The following steps explain the process in a stepwise manner:

- **Comparison of test case generation by constraint programming and metaheuristic approaches** – Before stepping into the combined approach of test case generation, we would like to compare the performance of metaheuristic and constraint programming to generate test cases in isolation from each other. The goal of executing this step is to find the direction of execution of the two solvers and this would be the baseline for our proposed approach. The solver (metaheuristic or constraint programming),which takes more time to find a solution would follow the other solver. The experiments performed on a sample code have been reported in Chapter 4.

- **Choice of the order of executing the metaheuristic and constraint programming solver** – As we are dealing with constraint programming and metaheuristic approaches, it is important to specify of direction of the flow of data from

one solver to the other. In the present scenario, we propose to start with the constraint programming solver, followed by the metaheuristic search engine. When dealing with input domains, it is better to provide a smaller domain for search to the metaheuristic engine, rather than constraint programming solver. From the experiment conducted (Section 4.1), we found out that metaheuristic approach takes more time to find a good solution compared to the constraint programming approach to reduce the search space. Thus, it is clear that metaheuristic approaches would benefit more compared to the constraint programming approaches if the domain to search is smaller (because it can search a solution faster in a smaller domain).

- **Constraint Programming Solver** – We split the individual variable input domains into their fractions by the use of dynamic domain splitting [3] approach. Dynamic domain splitting is a filtering technique applied on the numeric constraint satisfaction problem (CSP), where the domain of a variable is split in two, and the two resulting numeric CSPs are explored separately by backtracking. The procedure is based on a dynamic backtracking approach [37], which is briefly described. In a first step, the cause of a failure in the search is identified. Cause of the failure is the particular variable or the domain of a variable that has caused the search to block in some region of the search space. The cause is identified by deleting a set of values (based on the assignment statements) from each variable domain, thus introducing new domains for them (split domains for the variables). A failure would occur due to the most recent splitting constraint. To delete this splitting constraint, it is not enough to remove only the constraint, but also the effect it has on other variables and constraints. Decision constraints are also affected by this and the impact is more than that on any other variable. Therefore, all the statements in which the particular constraint (specially decision constraint) has been used would be identified and the original values would be restored. The process would continue with all the variables.

- **Reducing the Number of Input Variable Split Part Combinations** – Before reducing the split sub-domains of each variable, we would like to consider only the split sub-domains of each input variable that satisfies the constraint criterion(s) while trying to achieve a certain coverage. For example, consider there are three input variables x, y and z (each having a domain of 0-100), and a certain criterion states that for executing a particular branch, $x + y > z$. Each variable x, y and z will be split into four domains as: 0-25, 25-50, 50-75, 75-100. Now, out of the four sub-domains, if one of the split sub-domains of x is 0-25; 25-50 for y; and 75-100 for z, the criterion would never be satisfied, and thus there is no need

to consider this combination of sub-domain for further analysis. Checking these constraints, we can significantly reduce the number of split sub-domains for each.

Once done, we would split the selected sub-domains one more time, to have a knowledge of which part of the split sub-domain (first part or last part) can generate a better test case than the other. For example, if the split sub-domain $25 - 50$ of variable x satisfies the constraints for one particular test case, we would spilt this already split sub-domain into $25 - 37$ and $37 - 50$, to see further which of the two provides better test case. We know that it would lead to more computations, but this step is important to carry out iteratively to improve the test cases generated. Once we split the domains for the second time, we would reduce these domains and feed them to the metaheuristic solver to generate the test cases in a much reduced search space.

- **Reducing the input domains of the variables** - Following this is the reduction of specific sub-domains of the input variables into even smaller sub-domains using the VADA tool [36]. Considering the coverage criterion chosen to be branch coverage, the philosophy behind the reduction is to eliminate irrelevant input variables from the search space that does not have any role in reaching a particular branch or decision point in the control flow graph of the code. The reduction is done by forming slices for statements w.r.t. a variable. Slices are small broken pieces of code created for analysing them separately. For example, considering the running example of section 1.4, the predicate in statement 2 uses three variables a, b and c. The TRUE condition of this predicate uses the variable $NOT\_A\_TRIANGLE$ (statement 3). To justify a dependence relation between another variable $EQUILATERAL$ (statement 7) and $NOT\_A\_TRIANGLE$, the variable $NOT\_A\_TRIANGLE$ has to be sliced w.r.t. the variable $EQUILATERAL$. If the slice is empty, then the value of $EQUILATERAL$ does not depend on depend on either a, or b, or c (variables used in the predicate of statement 2). Each tuple consisting of reduced sub-domains from each variable is taken and fed into the metaheuristic solver.

- **Metaheuristic Solver** - Taking each pair of input tuples (split twice and reduced) from the database from the CP solver of input sub-domains of each variable, we would search for a local good solution (solution in the immediate neighbourhood of the search space concerned) and generate test case for the particular coverage criterion chosen. The proposed model then would then verify if a certain coverage criteria has been met or the whole search space has been traversed. If not, it would go back to the constraint programming solver and receive the next combination of input tuple from the CP solver and continue the process.

The example describes the above mentioned approach. Let x and y be the two input variables of the TRIANGLE example (Section 1.4). If the domain of x is set as $[0-1000]$, using step 1, it can be split into sub-domains like $[0-100], [100-200], [200-300], .., [900-1000]$. The choice of the number of splits and the range of each split is entirely problem dependent and would vary according to the problem. The extremes are included in the adjacent partitions to make sure that we do not have any good test case lying between the extremes like between 99 and 100. y can be similarly split into sub-domains. The next step would be to verify the constraint satisfaction criterion for each sub-domain of all the variables. Once the unwanted divided sub-domains are filtered, we would reduce each of the resulting sub-domains as mentioned before. This reduction is much less likely to miss a good solution (as compared to the already established approaches) because the input domain to reduce is only a fraction of the bigger domain.

We would then proceed further to take all the possible combinations of the input sub-domains, one by one, and feed them into the metaheuristic engine to search for the test cases. Thus, if we have n sub-domains for the variable x and m sub-domains for variable y after eliminating the unwanted sub-domains from x and y, we will have n $\times$ m number of combinations being fed into the metaheuristic search engine. The engine would take all the combinations one by one and generate test cases for each combination. The test cases that satisfies the coverage criteria would be then chosen. Iteratively the approach would go back to the constraint programming solver to further split the particular sub-domains of the specific test cases. Splitting at this step would allow us to explore which particular part of the sub-domain could generate better test cases than the other. Thus, the process is iterative in the sense that after each test case generation, the metaheuristic engine goes back to the constraint programming engine to receive the next input tuple (in terms of split domains) and generate a new set of test cases until the specific coverage criteria is reached. The process would stop once the coverage criteria has been met or no good solution has been found for the problem. In this way, we could explore the whole search space without overloading any of the solvers, because of the splitting and the reduction carried out before. The running example (Section 1.4) is executed in Chapter 4 and the pseudo code also detailed.

3.2 Verification of MPSoC Device with Software

Once we verify the platform on a prototype processor, we would check if it has been properly validated. If not, we would further apply stimuli generation, so as to generate new hard and soft constraints to verify the platform again. We would now look into the

issue regarding the compatibility of the software and the platform (Research Question 2). The key feature to the accomplishment of this step is the analysis of the abstraction of the system at various levels, *i.e.*, extracting abstractions at the component, functional and architectural level of the system. The reason for extracting abstraction at various levels is that abstraction allows separation of objects so that they can work on different levels of detail more easily. The abstraction is important considering the fact that we would require the knowledge of these abstractions to map them into constraint satisfaction problem of the software running on the platform. Further, we would like to generate proper test cases for specific test criterion to verify the system functionality. As a case study, we would be interested to run real world sample programs like DEMOSAIK and MPEG 4 coder on the MPSoC platform to verify the validity of the system as a whole. Figure 3.3 depicts the flowchart of the proposed approach.

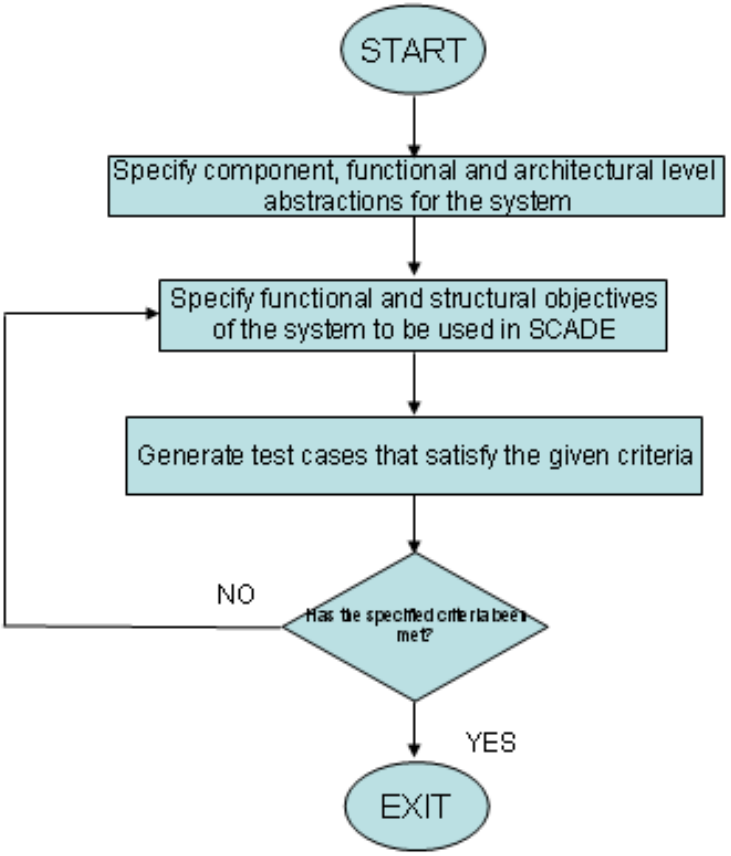

FIGURE 3.3: Verification of the Software running on the Hardware Platform

A detailed explanation of the approach is given in the following points:

- **Analysis of Various Abstraction Levels** – We think that dividing the software–hardware system into components, functions and architecture can help us to verify the hybrid system. Our intuition is that we can verify the hybrid system by

extracting abstractions for each of the component, functional and structural levels. The verification is a part of our long term goal and is explained in section 6.3. The first step to this approach would be to extract the abstractions of the various levels of the system. So, we would like to analyze the abstractions at the component level, functional level and architectural level.

  - **Component Level Abstraction** – Parsers have always been useful for extracting abstractions. We would develop parsers to exploit the abstractions to instrument the code of the MPSoC and application interaction.
  - **Functional Level Abstraction** – Control flow graph of the running software would be used to provide a guidance regarding the coverage criteria(s) to be used. Further, we would exploit the state diagram information from the functional specification to obtain the constraints to be used in the constraint satisfaction problem. Knowledge of these two parameters would enable us search area guidance, *i.e.*, knowledge of the areas where good solution can be found when we would generate test cases for the system verification.
  - **Architectural level abstraction** – As mentioned in Section 3.2, a layered architectural model would be exploited so that the abstractions can be extracted from each layer. We think extracting the abstractions from each layer would be easier because by this, we partition the system into layers and analyze the layers rather than abstracting the system as a whole.

- **Generation of Test Cases to Verify Specific Test Criterion(s) for the Hybrid System** - We would generate test cases for the system that will satisfy the coverage criterion and cause the system to fail. In this section, we propose a novel approach to do generate test cases. In this step, we would like to generate test cases automatically by the use of the functional and structural objectives. The following step elaborates the approach:

  - **Test Case Generation** - We would instrument a code that would generate test cases from the functional and structural test (assuming it to be available) requirements. As explained earlier in this chapter, we can generate test cases by the combination of metaheuristic and constraint programming approaches. The only difference is that here we would be dealing with the system, and not the software alone. Therefore, our structural test criteria would involve not only the coverage criteria like branch coverage but also a hardware requirement with it. For example, the criteria can be the coverage of a particular branch only when the hardware accelerator [5] is present in the MPSoC model. There can be several scenario(s) like this that would

make numerous system structural criteria(s). Once done, we would specify the structural test objectives for the system as a whole, which are mainly coverage criteria(s) like branch coverage or specific path coverage. We will also look into the possibility of using the SCADE tool [38] and its behaviour in accordance to the ReSP platform. The reason to look into the possiblity of using SCADE is because it is an efficient tool for analyzing embedded systems (Application software running of the MPSoC device) for test case generation. It integrates the Java-based Eclipse Applications Programming Interface (API) in compliance with the Eclipse Modeling Framework (EMF), and thus would be quite useful (test case generation on JAVA platform with SCADE interface) for the purpose of generating test cases for the hybrid software–hardware system. The tool works by generating a SCADE specification document from the requirement specification of the system to serve as the functional test objectives.

So, this chapter describes the whole approach needed to answer the two research questions mentioned in Chapter 1. Chapter 4 reports the preliminary results obtained while trying to answer Research Question 1.

Chapter 4

Preliminary Results and Ongoing Work

To provide answer to two research questions mentioned in Chapter 1, incremental steps need to be carried out. These steps are mentioned in Chapter 3. In this Chapter, we report the preliminary results obtained while conducting experiments regarding Research Question 1. Specifically, we want to solve the issue of the knowledge of the order of execution of the two solvers (metaheuristic and constraint programming), *i.e.*, which of the two solvers would follow the other, when we combine both the approaches.

4.1 Preliminary Results

The very first step that needs to be addressed to answer Research Question 1 is the comparison of the performance in terms of the execution times, number of branches covered and number of paths required for full branch coverage of metaheuristic and constraint programming approaches for the test case generation of a sample code. We run the experiment with our running example, the TRIANGLE code [1]. The relatively simple program takes three integer inputs and categorizes the triangles as equilateral, isosceles or scalene. For the comparison, we developed two programs, a metaheuristic (hill climbing) code and a constraint programming code to compare the execution times, number of branches covered and number of paths required for full branch coverage. We wanted to generate test cases by both the approaches separately for satisfying the branch coverage criterion of the sample code.

The sample code was instrumented and test input data generated using metaheuristics and constraint programming. The metaheuristics approach was run on Eclipse JVM

platform; while the constraint programming was executed by ILoG solver. As the computers used to run both metaheuristic and constraint programming approaches have the same configuration under the same load conditions, the results are comparable. Table 4.1 compares the performance of the two approaches.

Approach	Input range (For all variables)	Nbr. branches (in the code)	Nbr. branches (covered)	Execution time	Nbr. paths required (for full coverage)
CP	$0 - (2^{32} - 1)$	6	6	0.5 sec	4
Metaheuristic	$0 - (2^{32} - 1)$	6	6	2.6 sec	5

TABLE 4.1: Comparison of metaheuristic and constraint programming approaches on TRIANGLE code

It can be seen from the data in table 4.1 that while comparing the two approaches, constraint programming approach takes considerably less time (measured in seconds) than metaheuristic. It also takes lesser number of paths to cover all the branches. So, it's performance is superior to that of metaheuristic approach on this example. Further, it gave us an idea of the direction to choose, *i.e.*, which of the two solvers would follow the other solver. Metaheuristic takes more time to find a good solution, it would benefit more from the reduced domains of the variables, than constraint programming solver. Therefore, the constraint programming solver would be followed by the metaheuristic. We are in the process of carrying out more experiments to compare the performance of metaheuristics and constraint programming by instrumenting other sample codes. We think that the results of the experiments would make our point even stronger that constraint programming is superior to metaheuristics.

4.2 Ongoing Work: Explanation with A Running Example

Presently, our research work focusses on the implementation of the generation of test cases for our TRIANGLE running example. Now, we would like to explain with the same example and a pseudo-code, the combined metaheuristic-constraint programming approach for the generation of test cases for the software sub-system.

Consider the piece of code mentioned in Section 1.4 that takes three inputs a, b and c. Let us consider the specific path coverage as our criterion. Each statement in the sample code has been assigned (section 1.4) a specific number (eg. 1,2,3 as given on the left side of the code). In this example, we would like to reach the innermost branch, *i.e.*, statement 9 and so we choose our test path in the following order of statements: 1 – 2 – 4 – 5 – 6 – 8 – 9. The choice is obvious because we have to choose any one of the paths (as our test path) that leads the execution to the innermost branch of the code (most difficult to reach) and the path mentioned above is one of such paths. Therefore, our target is to generate a test case for this path. Following the steps mentioned in Section

3.1, the first step is the choice of the direction of the propagation of the approach. As mentioned in the previous section, we choose to start from the constraint programming approach and continue iteratively to the metaheuristic engine. The next step is to split the individual variable domains into parts (dynamic domain splitting), keeping a trade off between the number of splits and range of each split. We split each variable a, b and c of domains 0-100 each into four parts as the following sub-domains (SD): 0-25, 25-50, 50-75, and 75-100. The choice of the number of parts to be split is flexible and would depend on the problem. We have chosen four splits just as an example. Figure 5 shows the steps of splitting and reduction of the relevant sub-domains for one variable x. In the figure, the variable has been divided into four sub-domains each of range 25 (for example, 0-25, 25-50, etc.) and only sub-domain 1 and sub-domain 4 of variable x (SDx1 and SDx4) have been reduced, as only those sub-domains out of the total four satisfied the required constraints for the variable x. Again, just to explain the example in a simplified manner, we have chosen equal ranges for each of the four domains, *i.e.*, 25 each. A practical application may have variable domain ranges for each split. A similar logic would reduce the number of sub-domains for the other variables for further analysis. The approach is explained in the following paragraphs.

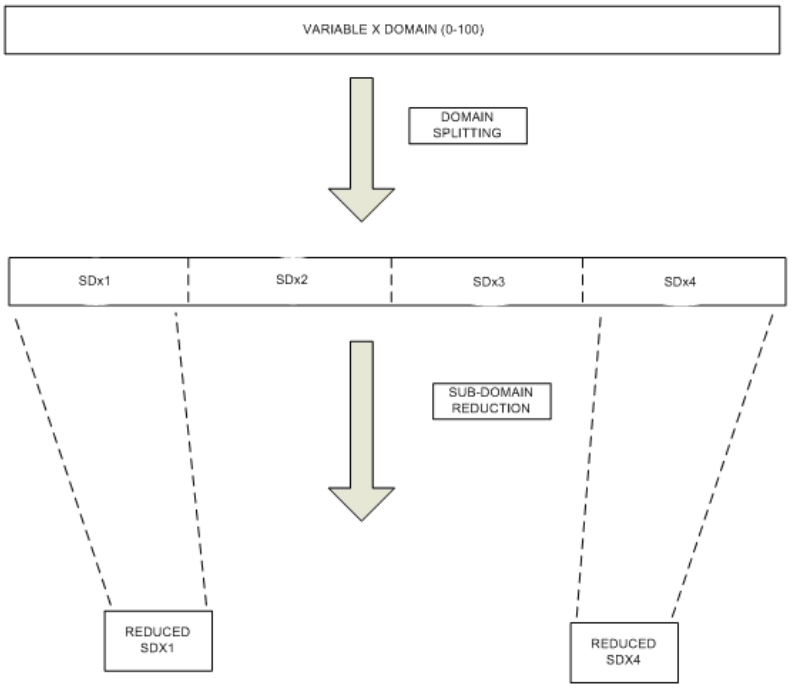

FIGURE 4.1: Input Variable Domain Splitting and Reduction Approach

Therefore we have a total of $4 \times 4 \times 4 = 64$ split sub-domain combinations or test cases possible. One test case comprises of one combination of split parts. We need to reduce to this number to decrease the overhead for the domain reduction step (VADA tool [36]). So, we would use the concept of constraint propagation to achieve this.

To cover the required test path mentioned before, we need to satisfy the constraint $(a + b > c)\&\&((a == b)\|(b == c))$.

Following provides all the possible combinations of test cases satisfying the constraint mentioned above. SDij stands for the jth split sub-domain of the variable i. The combination, for example, $\{SDx1, SDy1, SDz2\}$, depicts that the test case will consist one value between 0-25 for variable x, one value between 0-25 for value y, and one value between 25-50 for variable z. Out of the 64 test combinations, only the following 28 satisfy the above mentioned constraint,

$\{SDx1, SDy1, SDz1\}$, $\{SDx1, SDy1, SDz2\}$...$\{SDx2, SDy4, SDz4\}$ $\{SDx3, SDy4, SDz4\}$

So, we have obtained a reduction of 36 (from a total of 64 to only 28 test cases) test cases in this step. Now, we are in a position to reduce these 28 test case combinations using the VADA tool (shown by the dotted lines in Figure 3. Finally, once reduced, the sub-domains are fed into the metaheuristic engine to generate a test case from the smaller domains. To cover another path in the next iteration, the model would first check if the coverage criteria (branch coverage and specific path coverage) has been met or the search space has been explored. If neither is true, it would go back to the constraint programming engine to generate new set of input sub-domains to be fed into the metaheuristic engine to generate new test cases; else it would terminate the process because the goal for coverage criteria has been reached. The pseudo code for the above mentioned approach is as follows:

Algorithm 1 Software Verification

```
1:  L ← EmptyList{Clusters}
2:  while all the input sub-domain combinations have been executed do
3:    while all the input variables have been split do
4:      while whole domain of one variable has been partitioned do
5:        create subdomains of the variable.
6:      end while
7:    end while
8:
9:    while all possible test combinations involving the sub-domains are checked do
10:     constraint propagation check for each sub-domain for one criteria.
11:   end while
12: end while
13: while the relevant input variable sub-domains have been reduced do
14:   create reduction for specific the sub-domains.
15:   feed input tuple into metaheuristic engine.
16:   generate test case.
17: end while
18: if coverage criteria has meet met OR search space has been explored then
19:   EXIT.
20: else
21:   Check for the best test case and split the sub-domain of this test case to feed into metaheuristic solver to
      continue the process.
22: end if
```

Presently, we are in the process of implementing the proposed approach for the software verification and various results associated with the experiments will be reported soon.

Our starting point is the same TRIANGLE sample code [1] and we are trying to generate test cases for this code using our proposed approach. We have already compared the performance of constraint programming and metaheuristic approaches on the same code. Further, as mentioned in Chapter 3, we are also working on the improvement of a solution generated by the approach based on the direction of movement.

Chapter 5

Conclusion

We have proposed a research plan to answer some of the research issues related to the verification of the software and the system consisting of a software running on a multiprocessor hardware platform. Research issues are still open regarding the verification of the software, the multiprocessor platform (ReSP) and the hardware-software interface. We therefore framed the research questions accordingly as:

- How can we verify the software portion of the MPSoC device so as to overcome the limitations (scalability issue for constraint programming, problem of getting confined to the local optimum for metaheuristic approaches and loss of good test cases when combining the two approaches) of the established approaches and satisfy specific coverage criteria(s) to generate cases leading to the system failure?
- How can we generate test cases for checking the compatibility of software running on the MPSoC platform, thereby verifying the functional and structural coverage criteria(s) of the system as a whole?

Out expected research contributions can be summarized as follows:

- Generation of test cases for a software system by the combination of metaheuristic and constraint programming approaches, to satisfy specific coverage criterion and raising exceptions leading to the software failure. The research works carried out in the domain of test case generation of software have specific limitations: Constraint programming approaches do not perform well when used in complex and big programs (they did perform well, as explained in chapter 4, because the program under test was a small and simple one); metaheuristic approaches gets confined in the local optimum and thus does not find good solutions in the search space;

combination of constraint programming and metaheuristic approaches results in loss of many good solutions because they rely on reducing the individual input variable domains. Our objective is to overcome the the mentioned limitations by building on the established approaches [22].

- Satisfying the structural and functional criteria(s) of the system as a whole (comprising of the software running on the hardware platform), while exploiting the various levels of abstractions of the system. The verification is done by developing a code that would generate test cases from the functional test objectives. Specifying the structural test objectives for the system, like coverage criteria(s) like branch coverage or specific path coverage,test cases are generated by the combination of metaheuristic and constraint programming approaches. We will deal with the system and not the software alone. Our structural test criteria would involve not only the software requirement like branch coverage, but also a hardware requirement with it.

- Verification of the overall system by applying the approach on the DEMOSAIK and MPEG 4 codes running on the MPSoC platform.

We propose research directions to answer each of the two research questions. Considering the problem as consisting of multiple parts, we frame sequential approaches in a stepwise manner to reach the solution of the questions. The proposed approaches are expected to overcome the limitations of the existing methods. Experiments were carried out to answer a part of the first research question and the results provided a guidance (direction of use of the two solvers sequentially) for the next step.

Presently, we are working on exploring the validation of the first step and subsequent steps of the first research problem, while simultaneously carrying out experiments with the a sample application running on the ReSp platform.

Chapter 6

Research Goals and Timeline

Chapter 1 mention the goals of our research work. Two research questions were addressed to achieve the goals. In this chapter, we will mention the short term, medium term, and long term goals of our research. Further, the chapter will provide the timeline, *i.e.*, the work and duration in which we would expect to achieve the research goals, along with the papers that we would expect to publish in various international conferences and journals. Following sections provide the research goals and the timeline.

6.1 Short Term Goals

- Comparing the performance of constraint programming and metaheuristic approaches on a more complex sample code (the comparison has been carried out on the TRIANGLE sample code as in Section 1.4) like DEMOSAIK or MPEG 4 to generate test cases by both the approaches so as to cause the program to fail. The performance is the baseline of our research as it will provide the order in which two solvers (metaheuristic and constraint programming) will be executed to generate test cases for the software.
- Combining metaheuristic and constraint programming approach (Section 3.1) by splitting the input variable domains in the metaheuristic solver, checking the constraint satisfaction and reducing the only relevant input sub-domains to be fed into the metaheuristic solver to generate test cases iteratively. The generation will stop once the coverage criteria has been met or it is found that no solution exists for the criteria.

We expect to address the short term goals and obtain the results by Winter'2011 or early Summer'2011, and publish technical papers in ICST 2011 and ISSTA 2011.

6.2 Medium Term Goals

- Analysis of the system to be broken down into component and functional levels.
- Extraction of the abstractions at the component, architectural and functional level of the system consisting of the application software running on the hardware MP-SoC platform.

We hope that we will answer the medium term goals by the end of Winter'2012 or early Summer'2012, and expect to publish technical papers in TSE and TOSEM.

6.3 Long term goals

- Generating test cases for the hybrid system to satisfy specific coverage criteria and cause the system to fail. The test case generation generation will comprise of using the functional and structural requirements of the system.

By the early Summer'2013, we expect to address the long term goals, and publish technical papers in EMSE and ICSM'2013.

6.4 Research Timeline

	F10	W11	S11	F11	W12	S12	F12	W13	S13	Publications
Course-Work	x	x								
RQ1	x	x	x	x						ICST'11, ISSTA'11
RQ2			x	x	x	x				TSE, TOSEM, EMSE
Thesis Writing							x	x		
Defence									x	

TABLE 6.1: Research Timeline

RQ1: Verification of software; RQ2: Verification of the software-hardware hybrid system

Bibliography

[1] Phil Mcminn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[2] Roger Pressman. Software engineering: A practitioner's approach, 3rd edition. *McGraw Hill*, 1992.

[3] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. In *IAAI'06: Proceedings of the 18th conference on Innovative applications of artificial intelligence*, pages 1720–1727. AAAI Press, 2006. ISBN 978-1-57735-281-5.

[4] David Patterson. The trouble with multicore. *IEEE Spectrum*, 2010.

[5] Pieter J. Mosterman Gabriela Nicolescu. Model-based design for embedded systems. 2009.

[6] Shuichi Fukuda. Complex systems concurrent engineering.

[7] N. Tracey, J. Clark, and J. Mcdermid. Automated test-data generation for exception conditions. *Software - Practice and Experience*, 30:61–79, 2000.

[8] Mark Harman. Open problems in testability transformation, 2008.

[9] Steven Prestwich. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115:51–72, 2002.

[10] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 169–174. AAAI Press, 2000. ISBN 0-262-51112-6.

[11] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201157675.

[12] Talbi E.G. Metaheuristics: from design to implementation. 2009.

[13] Krzysztof Apt. Principles of constraint programming. 2003.

[14] Eyun Eli Jacobsen. Design patterns as program extracts. *Lecture notes in Computer Science*, 1998.

[15] Xue-mei Yan; Wei-jiang Wang, Ting-zhi Shen. An improved alternating-projection demosaicing algorithm. *CIS'08*, 2008.

[16] Joachim Wegener Harmen Sthamer, Andr Baresel. Evolutionary testing of embedded systems. *14th International Internet and Software Quality Week*, 2001.

[17] Phill McMinn. How does program structure impact the effectiveness of the crossover operator in evolutionary testing? *2nd International Symposium on Search Based Software Engineering*, 2010.

[18] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: http://doi.acm.org/10.1145/1287624.1287647.

[19] Philippe Refalo. Impact-based search strategies for constraint programming. *International Conference on Principles of Constraint Programming 2004*, 2004.

[20] Lus Paulo Reis and Eugnio Oliveira. A constraint logic programming approach to examination scheduling. *Foundation for Science and Technology.*, 2008.

[21] Francois Laburthe A. L. Filippo Focacci. Constraint and integer programming: Toward a unified methodology. *Operations Research/Computer Science Interfaces Ser.*

[22] Mara Cristina Riff Carlos Castro, Michael Moossen. A cooperative framework based on local search and constraint programming for solving discrete global optimisation. *SBIA, Lecture Notes in Computer Science*, 2007.

[23] Kim-Hoa Lam Tuan-Anh Duong. Combining constraint programming and simulated annealing on university exam timetabling, 2004.

[24] Richard Goering. Networking concepts inspire next-gen socs. *Electronic Engineering Times*, 2005.

[25] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers and Operations Research*, 2007.

[26] I. Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. *Fromman-Holzboog Verlag, Stuttgart*, 1973.

[27] J. H. Holland. Adaptation in natural and artificial systems. *The University of Michigan Press, Ann Arbor, Michigan*, 1975.

[28] Ruilian Zhao. Character string predicate based automatic software test data generation, 2003.

[29] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *In: Proc. European Dependable Computing Conference. Volume 3463 of LNCS (2005) 281292*, pages 281–292. Springer. ISBN, 2005.

[30] Fabrizio Ferrandi, Michele Rendine, and Politecnico Di Milano. Functional verification for systemc descriptions using constraint solving. In *In Design, Automation and Test in Europe*, pages 744–751, 2002.

[31] Roy Emek, Itai Jaeger, Yehuda Naveh, Gadi Bergman, Guy Aloni, Yoav Katz, Monica Farkash, Igor Dozoretz, and Alex Goldin. X-gen: A random test-case generator for systems and socs. In *In IEEE International High Level Design Validation and Test Workshop*, pages 145–150, 2002.

[32] Bruno De Backer, Vincent Furnon, P. Kilby, P. Prosser, and P. Shaw. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6:501–523, 1997.

[33] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. Introduction to algorithms, second edition.

[34] Koushik Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321746.

[35] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. Technical report, 2008.

[36] Youssef Hassoun Chris Fox, Mark Harman. Variable dependence analysis technical report: Tr-10-05. 2010.

[37] Roie Zivan and Amnon Meisels. Concurrent dynamic backtracking for distributed csps. In *In CP-2004*, pages 782–787, 2004.

[38] Christel Seguin Virginie Wiels Guy Durrieu, Odile Laurent. Automatic test case generation for critical embedded systems, 2002.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL