Chapter 1

Introduction

Embedded systems (ES) are information processing systems that are embedded into different products. These systems are dedicated towards certain applications with real-time constraints, reactive in nature, and must be dependable and efficient (Marwedel, 2006). Embedded systems pervade all aspects of modern life, spanning from consumer electronics, household appliances, automotive electronics, aircraft electronics and telecommunication systems. These systems are also suited for use in transportation, safety and security, robotics, medical applications and life critical systems. In the subsequent sections, we describe the ES design flow and highlight various behavioural transportation that are often applied in course of the design. Since dependability is an important aspect of ES, we have taken up the issue of verifying the correctness of such transformations by way of equivalence checking. Specific problems handled in the thesis have been listed. This is followed by a statement of contributions and the thesis organization.

1.1 Embedded system design flow

Present day electronic products demand high performance and extensive features. As a consequence, embedded systems are becoming more complex and difficult to design. To combat complexity and explore the design space effectively, it is necessary to represent systems at multiple levels of abstraction (Chen et al., 2006b). Initial functions and architectures are preferably specified at a high level of abstraction. Functions

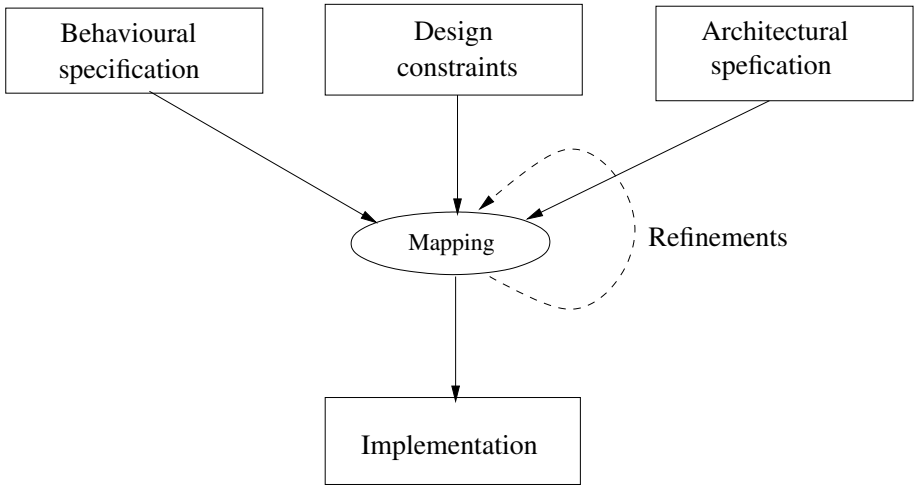

Figure 1.1: Embedded system design flow

are then mapped onto the architecture through an iterative refinement process (Chen et al., 2003; Keutzer et al., 2000). The embedded system design flow based on this principle is shown in figure 1.1. During the refinement process, the initial design is repeatedly partitioned into hardware (HW) and software (SW) parts – the software part executes on (multi)processor systems and the hardware part runs as circuits on some IC fabric like an ASIC or FPGA. Application-specific hardware is usually much faster than software, but its design, validation and fabrication are significantly more expensive. Software, on the other hand, is cheaper to create and to maintain, but it is slow. Therefore, the performance-critical components of the system should be realized in hardware, and non-critical components should be realized in software. The HW-SW partitioning approach is shown in figure 1.2.

The SW part of the behaviour is translated into assembly/machine language code by compilers. The translation process of input behaviour into target assembly language code can be divided into two stages: the front end for analysis and the back end for synthesis (Raghavan, 2010). The front end of the compiler transforms the input behaviour into an intermediate representation (IR) and then the back end transforms the IRs into a target assembly language code. The front end consists of several subtasks such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation and optimizations (Aho et al., 1987; Raghavan, 2010). The back end of the compiler consists of target code generation and optimization tasks. A set of behavioural transformations are applied in the optimization phase by modern compilers during this translation processes (Muchnick, 1997). Figure 1.2 highlights the list of

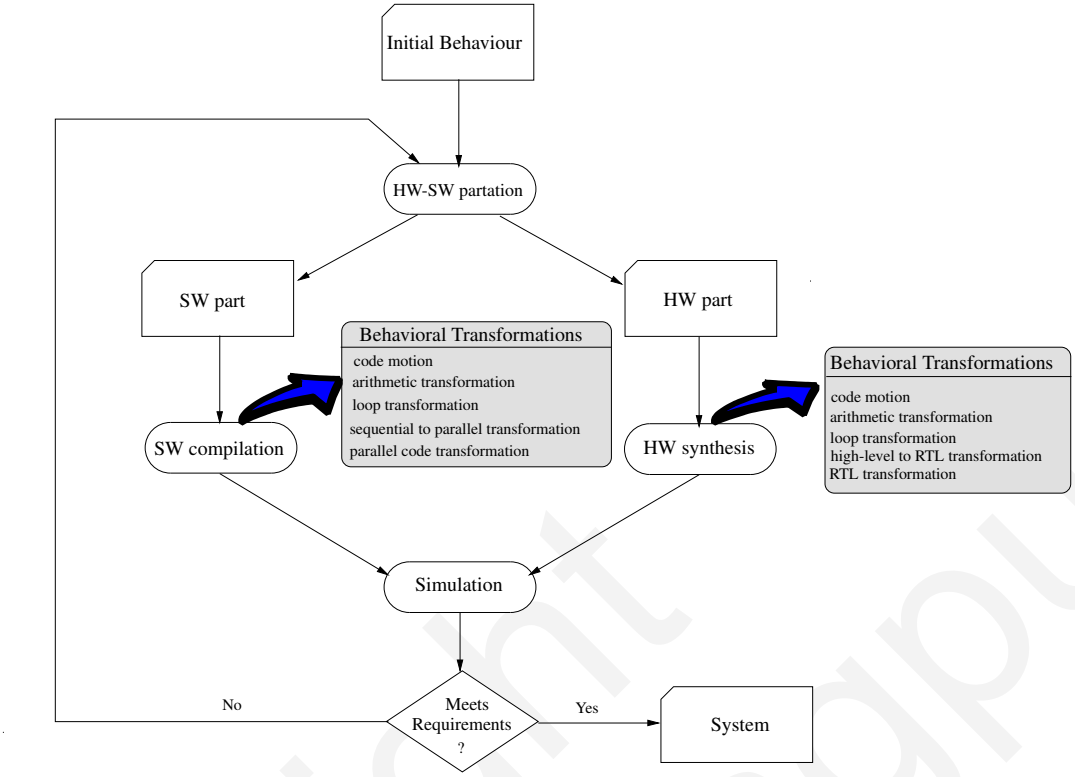

Figure 1.2: HW-SW partitioning

behavioural transformations that are applied during SW-compilation processes.

The HW part of the behaviour is first translated into a register transfer level (RTL) code by high level synthesis (HLS) tools (Gajski et al., 1992; Gupta et al., 2003c). The generated RTL then goes through the logic synthesis and the physical design process before being fabricated into a chip. The HLS process consists of several interdependent subtasks such as compilation or pre-processing, scheduling, allocation and binding, and datapath and controller generation. The HLS process applies several behavioural transformations during pre-processing and scheduling phases. The list of behavioural transformations that are commonly used during these phases are listed in figure 1.2. In the next section, we discuss the commonly used behavioural transformations during embedded system synthesis.

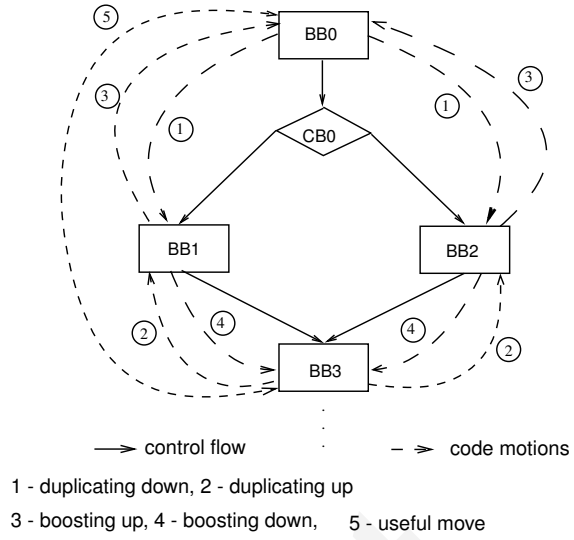

Figure 1.3: Various code motion techniques

1.2 Behavioural transformations

As discussed above, application of behavioural transformation techniques is a common practice during embedded system design. In course of devising the final implementation from the initial specification, a set of transformations may be carried out on the input behaviour targeting the optimal performance, energy and/or area on a given platform. In this section, we introduce several behavioural transformations that are commonly applied during embedded system design.

1.2.1 Code motion transformations

Code motion is a technique to improve the efficiency of a program by avoiding unnecessary re-computations (Knoop et al., 1992). The primary objective of code motion is the reduction of the number of computations at run-time. Secondary objective is the minimization of the lifetimes of temporary variables to avoid unnecessary register pressure. This can be achieved by moving operations beyond basic block boundaries. The code motion based transformation techniques can be classified into the following categories (Rim et al., 1995) using Fig 1.3: (*i*) Duplicating down refers to moving operations from a basic-block (BB) preceding a conditional block (CB) to both the BBs following the CB. *Reverse speculation* (Gupta et al., 2004b) and *lazy execution* (Rim

et al., 1995) belong to this category. (*ii*) Duplicating up involves moving operations from a BB in which conditional branches merge to its preceding BBs in the conditional branches. *Conditional speculation* (Gupta et al., 2004b) and *branch balancing* (Gupta et al., 2003a) fall under this category. (*iii*) Boosting up moves operations from a BB within a conditional branch to the BB preceding the CB from which the conditional branch sprouts. Code motion techniques such as *speculation* (Gupta et al., 2004b) fall under this category. (*iv*) Boosting down moves operations from BBs within the conditional branches to a BB following the merging of the conditional branches (Rim et al., 1995). (*v*) Useful move refers to moving an operation to a control and data equivalent block. A code motion is said to be *non-uniform* when an operation moves from a BB preceding a CB to only one of the conditional branches, or vice-versa. Conversely, a code motion is said to be *uniform* when an operation moves to and fro both the conditional branches from a BB before or after the CB. Therefore, duplicating up and boosting down are inherently uniform code motions whereas *duplicating down and boosting up can be uniform as well as non-uniform*.

1.2.2 Loop transformations

As the name suggests, loop transformation techniques are used to increase instruction level parallelism, improve data locality and reduce overheads associated with executing loops of array-intensive applications (Bacon et al., 1994). Most execution time of a scientific program is spent on loops. Thus, a lot of compiler analysis and compiler optimization techniques have been developed to make the execution of loops faster. *Loop fission/distribution/splitting* attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop body. The inverse transformation of loop fission is *fusion/jamming*. *Loop unrolling* replicates the body of a loop by some number of times (unrolling factor). Unrolling improves performance by reducing the number of times the loop condition is tested and by increasing instruction parallelism. *Loop skewing* takes a nested loop iterating over a multi-dimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop. *Loop interchange/permutation* exchanges inner loops with outer loops. Such a transformation can improve locality of reference, depending on the array layout. *Loop tiling/blocking* reorganizes a loop to iterate over blocks of data sized to fit in the cache.

Loop unswitching moves a conditional from inside a loop to outside by duplicating the loop body. Some other important loop transformations are loop *reversal, spreading, peeling*, etc. (Bacon et al., 1994).

1.2.3 Arithmetic transformations

A compiler usually applies a set of techniques that transform the arithmetic expressions of the behaviours. Some of them are discussed here. *Common subexpression elimination:* In many cases, a set of computations will contain identical subexpressions. The compiler can compute the value of the subexpression once, store it, and reuse the stored result. *Constant propagation*: Typically programs contain many constants. By propagating them through the program, the compiler can do a significant amount of precomputation (Bacon et al., 1994). More importantly, the propagation reveals many opportunities for other optimization. *Constant folding* is a companion to constant propagation. When an expression contains an operation with constant values as operands, the compiler can replace the expression with the result. *Copy propagation*: Optimization such as common subexpression elimination may cause the same value to be copied several times. The compiler can propagate the original value through a variable and eliminate redundant copies. *Algebraic transformations:* The compiler can simplify arithmetic expressions by applying *algebraic rules such as associativity, commutativity and distributivity*. *Operator strength reduction:* The compiler can replace an expensive operator with an equivalent, less expensive operator. *Redundancy-eliminating transformations*: Compiler may remove unreachable and useless computations. A computation is unreachable if it is never executed. Removing it from the program will have no semantic effect on the instructions executed. A computation is useless if none of the outputs of the program are dependent on it.

1.2.4 High-level to RTL transformations

High-level synthesis (HLS) tools (Gajski et al., 1992) convert a high-level behavioural specification into an RTL level description. The HLS process consists of several interdependent subtasks such as compilation or pre-processing, scheduling, allocation and binding, and datapath and controller generation. In the first step of HLS, the be-

havioural description is compiled into an internal representation. This process usually includes a series of compiler like optimizations. Scheduling assigns operations of the behavioural description into control steps. Allocation chooses functional units and storage elements from the component library based on the design constraints. Binding assigns operations to functional units, variables to storage elements and data transfers to wires or buses such that data can be correctly moved around according to the scheduling. The final step of high-level synthesis is data-path and controller generation. Depending upon the scheduling and the binding information of the operations and the variables, proper interconnection between the data-path components is set up. Finally, a finite state machine (FSM) is generated to control all the micro-operations over the datapath. The RTL designs consist of a description of the datapath netlist and a controller FSM.

Power optimization can be performed on different levels of the design hierarchy. Lately, system level and high-level power optimization have received great attention (Ahuja et al., 2010; Jiong et al., 2004; Lakshminarayana et al., 1999; Musoll and Cortadella, 1995; Xing and Jong, 2007). Employing low power transformations at RTL designs, such as, restructuring multiplexer networks (to enhance data correlations and eliminate glitchy control signals), clocking control signals, and inserting selective rising/falling delays, clock gating, etc., has emerged as an important technique to minimize total power consumption of the circuits (Chandrakasan et al., 1995a, 1992; Raghunathan et al., 1999). In control-flow intensive designs, the multiplexer networks and registers dominate the total circuit power consumption. Also, the control logic can generate a significant amount of glitches at its outputs, which in turn propagate through the data path accounting for a large portion of the glitch power in the entire circuit. For data-flow intensive designs, the chaining of arithmetic functional units results in majority of dynamic power consumption.

1.2.5 Sequential to parallel transformations

Applications like multimedia, imaging, bioinformatics, and signal processing must achieve a high computational power with minimal energy consumption. Given these conflicting constraints, multiprocessor implementations not only deliver the necessary computational power, but also provide the required power efficiency. However, the performance gain achieved is dependent on how well the compiler can parallelize the

given program and generate code for ‘matching’ the hardware parallelism. There are three types of parallelism of the sequential programs: (i) Loop-level parallelism: The iterations of a loop are distributed over multiple processors. (ii) Data-parallelism: data parallelism is achieved when each processor performs the same task on different pieces of distributed data. (iii) Task-level parallelism: Task parallelism focuses on distributing sub-tasks of a program across different parallel processors. The sub-tasks can be the subroutine calls, independent loops, or even independent basic blocks. This is the most used parallelization technique.

The parallel behaviour obtained from a sequential behaviour may undergo a parallel level code transformations. Parallel transformations manipulate the concurrency level of the parallel programs. The concurrency level of a program may not match the parallelism of the hardware, or vice-versa. In this case, the performance (in terms of total execution time, energy consumption, etc.) of that parallel program can be modified to suit the hardware by changing the concurrency level of the program. The transformations like *channel merging and splitting, process merging and splitting and computation migration, etc.*, are commonly used for this purpose.

1.3 Motivations and objectives

Verifying correctness of behavioural transformations, as described above, is an important step in ensuring dependability of embedded systems. Various analysis tools can prove the absence of certain kinds of errors at the source behaviour; however, if the compiler is not guaranteed to be correct, then no source-level guarantees can be safely transferred to the generated code. One of the most error prone parts of a compiler is its optimization phase. Many optimizations require an intricate sequence of complex transformations. Often these transformations interact in unexpected ways, leading to a combinatorial explosion in the number of cases that must be considered to ensure that the optimization phase is correct (Kundu et al., 2009). Vendors of mature ES design tools often talk of method which are “correct by construction". However, it should be noted that tools are always in a state of change as newer features have to be introduced to maintain competitiveness of the tools. Therefore, even with “mature" tools, there is a possibility of encountering a bug resulting in design flaws.

A degree of assurance is achieved by way of on extensive simulation and testing. The goal of simulation is to identify errors as early as possible in the design phase. In particular, for embedded systems, both the hardware and the software parts of the system must be simulated at the same time. Both simulation and testing suffer from being incomplete: each simulation run or each test evaluates the system performance for only a single set of operating conditions and input signals. For complex embedded systems, it is impossible to cover even a small fraction of the total operating space with simulations. Finally, testing is prohibitively expensive. Today building a test harness to simulate a component's environment is more expensive than building the component itself.

Formal verification can be used to provide guarantees of compiler correctness. It is an attractive alternative to traditional methods of testing and simulation, which for embedded systems, tend to be expensive, time consuming, and hopelessly inadequate, as argued above. There are two fundamental approaches of formal verification of compilers. The first approach proves that the steps of the compiler are *correct by construction*. In this setting, to prove that an optimization is correct, one must prove that for any input program the optimization produces a semantically equivalent program. The primary advantage of correct by construction techniques is that optimizations are known to be correct when the compiler is built, before they are run even once. Most of the techniques that provide correct by construction guarantees require user interaction (Kundu et al., 2009). Moreover, correct by construction proofs are harder to achieve because they must show that *any* application of the optimization is correct. Despite the fact that a significant amount of work has been carried out for verifying that synthesis steps are correct by construction, the state of the art techniques are still far from being able to prove automatically that the steps of the compilation process always produce correct designs (Kundu et al., 2010). However, even if one cannot prove a compiler to be correct by construction, one can at least show that, for each translation that a compiler performs, the output produced has the same behavior as the original behaviour. The second category of formal verification approach, called *translation validation*, consists of proving correctness each time an optimization step is invoked. Here, each time the compiler runs an optimization, an automated tool tries to prove that the original program and the corresponding optimized program are equivalent. Although this approach does not guarantee the correctness of the compilation process, it at least guarantees that any errors in translation will be caught when the

particular steps of ES design tool are performed, preventing such errors from propagating any further in synthesis process. In this dissertation, we work on developing translation validation methodologies for several behavioural transformations applied during embedded systems.

1.3.1 Problem statements

The objective of this work is to show the correctness of several behavioural transformations that occur during embedded system design primarily using equivalence checking methods. Specifically, the following verification problems will be addressed:

Code motion transformations: Several code motion techniques such as speculation, reverse speculation, branch balancing, conditional speculation, etc., may be applied on the input behaviour which is in the form of a sequential code at the preprocessing stage of embedded system synthesis. The input behaviours are transformed significantly due to these transformations. One may also apply a combination of these transformation techniques. Moreover, arithmetic transformations may also be applied along with code motion transformations. It is, therefore, a non-trivial task to show the equivalence between the input behaviour and the transformed behaviour. The equivalence checking methods reported in the literature conventionally use path based approach whereupon for each path in a behaviour an equivalent path is identified in the transformed behaviour. Code motions can be of two types namely, uniform and non-uniform code motions. The former moves a code segment over both branches following a branching block; in contrast, the latter category of code motions move code segments over only one branch of the BB. The methods reported in the literature can verify only uniform code motions. A common verification mechanism for both kinds of code motion transformations is worth exploring.

High-level to RTL transformations: During high-level synthesis, the input high-level behaviour is transformed to an output RTL consisting of a datapath, which is merely a structural description, and a controller, represented as a finite state machine (FSM). The controller invokes a control assertion pattern (CAP) in each control step to execute all the required data-transfers and proper operations in the FUs. The results of the relational operations (i.e. the status signals) are the inputs to the controller. The state transitions in the controller FSM depend on these status signals. The input be-

haviour is in a higher abstraction level compared to that of the output RTL behaviour. The verification method, therefore, should first analyze the CAP vis-a-vis the datapath structure to identify the RT-operations from the output RTL behaviour to compare it with the input high-level behaviour. The data transformations of the high-level behaviour, however, may be implemented in pipelined or multicycle functional units which may execute over more than one FSM control state. Therefore, a state-wise analysis of CAP vis-a-vis datapath inter-connections may not suffice to verify multicycle and pipelined operations. The verification task of this phase, therefore, should accommodate all the intricacies of the RTL designs to show the equivalence between the high-level behaviour and the RTL behaviour.

RTL transformations: The low power RTL transformations primarily bring about modifications at a much lower level of abstraction involving intricate details regarding restructuring of the datapaths, control logic and routing of the control signals. Accordingly, in a typical industry scenario, an RTL or architectural low power transformation implies a full cost of simulation based validation, which can extend to many months (Viswanath et al., 2009). Formal verification methods, not necessarily compromising the details through abstraction, is a desirable goal. One of the objectives of this work is showing equivalence of two RTL designs, one obtained from the other by applying low power RTL transformations.

Loop transformations: Signal processing and multimedia applications are data dominated in nature. Several loop based transformation techniques such as un-switching, reordering, skewing, tiling, unrolling, etc., may be applied at the preprocessing stage. These transformations are applied in order to reorder and restructure the loop body to improve the spatial and temporal locality of the accessed data. The loop transformation techniques can be of two types, structure preserving and structure modifying. The former admits a clear mapping of control and data values in the transformed behaviour to the corresponding control and data values in the source behaviour; the latter does not admit such a mapping (Zuck et al., 2005). In addition, arithmetic transformations may also be applied along with loop transformations. The verification method should be strong enough to handle as many loop transformations and arithmetic transformations as possible applied on data dominated behaviours.

Sequential to parallel behaviour transformation: The functional specification, which relies on a single-threaded sequential code, is not easy to deploy on highly

concurrent heterogeneous multi-processor systems. A parallel model of computation (MoC) may be used as the programming model. However, writing an application depicting concurrency is time consuming and error prone which conflicts with the low time-to-market requirement of the present day embedded systems. So, an automated tool that converts a sequential code to its equivalent concurrent behaviour is employed. In the case of streaming applications, Kahn process network (KPN) (Kahn, 1974) model of computation is often used. The KPN is a deterministic parallel MoC that explicitly specifies tasks as processes and distributed memory units as FIFO channels. The verification task involves showing the equivalence between a sequential behaviour and its corresponding parallel KPN behaviour. Obviously, the overall data transformation of the KPN behaviour can only be captured if the global data dependencies are captured. The challenge lies in capturing the data dependencies spread over all the concurrent KPN processes independent of their possible interleaving.

Parallel level transformations: The parallel process network model that are obtained from the sequential behaviour in an automated way or manually may not be suitable for the target architectural platform. Moreover, the overall achieved performance obtained as a result of mapping may not be satisfactory from the point of view of the data throughput and/or memory usage. In this case, it is necessary to manipulate the amount of concurrency in the functional model. Too little concurrency may not be desirable since it may not completely make use of the architectural capabilities. Too much concurrency, in contrast, may require too large an overhead to eventually manage in the architecture, which may increase the overall latency. The transformation techniques like process splitting, channel merging, process clustering, and unfolding and skewing may be used to control the concurrency in the KPN behaviour according to the architectural constraints. The verification task of this phase, therefore, is to show the equivalence between two KPN behaviours. The challenge remains the same as in equivalence checking of sequential to parallel transformation namely, capturing the global dependencies irrespective of the interleaving of both the input and the output concurrent behaviours. Hence, it is worth examining whether a method capable of validating sequential to parallel transformations should suffice for the commonly used parallel level transformations.

1.4 Contributions

Verification of code motion transformations: A formal verification method for checking correctness of code motion techniques is presented. Finite state machine models with datapath (FSMDs) have been used to represent the input and the output behaviours. The method introduces cutpoints in one FSMD, visualizes its computations as concatenation of paths from cutpoints to cutpoints, and then identifies equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. A path is extended when its equivalent path cannot be found in the other FSMD. However, a path cannot be extended beyond loop boundaries. Our method is capable of verifying both uniform and non-uniform code motion techniques. It has been underlined in this work that for non-uniform code motions, identifying equivalent path segments involves model checking of some data-flow properties. Our method automatically identifies the situations where such properties are needed to be checked during equivalence checking, generates the appropriate properties, and invokes the model checking tool NuSMV to verify them. The correctness and the complexity of the method have been dealt with. Experimental results demonstrate the effectiveness of the method.

Verification of RTL generation and RTL transformations: A formal verification method of the RTL generation phase of a high-level synthesis (HLS) process is presented in (Karfa, 2007). The goal is achieved in two steps. In the first step, the datapath interconnection and the controller FSM description generated by a high-level synthesis process are analyzed to obtain the register transfer (RT) operations executed in the datapath for a given control assertion pattern in each control step. In the second step, an equivalence checking method is deployed to establish equivalence between the input and the output behaviours of this phase. A rewriting method has been developed for the first step. Our method is strong enough to handle pipelined and multicyle operations, if any, spanning over several states. The correctness (termination, soundness and completeness) and complexity of the presented method have been treated formally. The experimental results on several HLS benchmarks are presented.

In order to verify RTL transformations, we model both the RTLs as FSMDs and then apply our FSMD based equivalence checking method to show the equivalence. In this work, we analyze several commonly used RTL low power transformation tech-

niques and show that our verification method can handle those transformations.

Verification of loop and arithmetic transformations of array intensive behaviours: We propose a formal verification method for checking correctness of loop transformations and arithmetic transformations applied on the array and loop intensive behaviours in design of area/energy efficient systems in the domain of multimedia and signal processing applications. Loop transformations hinge more on data dependence and index space of the arrays than on control flow of the behaviour. Hence, array data dependence graphs (ADDGs), proposed by Shashidhar (Shashidhar, 2008), are used for representing array intensive behaviours. Shashidhar (Shashidhar, 2008) proposed an ADDG based equivalence checking method to validate the loop transformations. Possible enhancements of his method to handle associative and commutative transformations are also discussed in (Shashidhar, 2008; Shashidhar et al., 2005a). We redefine the equivalence of ADDGs in this work to verify loop transformations along with a wide range of arithmetic transformations. We also propose a normalization method for array intensive integer arithmetic expressions. The equivalence checking method relies on this normalization technique and some simplification rules to handle arithmetic transformations over arrays. Correctness and complexity of the method have been dealt with. Experimental results on several test cases are presented.

Verification of Parallelizing transformations: A formal verification method for checking correctness of parallelizing transformations of sequential behaviours is presented. The parallel behaviours are represented as Kahn process networks (KPNs). We describe a mechanism to represent a KPN behaviour comprising inherently parallel processes as an ADDG. We also present a proof of correctness of the construction mechanism. We then apply our ADDG based equivalence checking method to establish the equivalence between the initial sequential behaviour and the KPN behaviour. The key aspect of our scheme is to model a KPN behaviour as an ADDG. We then show how the ADDG based modelling helps us detect deadlocks in KPN behaviours. Experimental results supporting this scheme are provided.

We next address the problem of verifying transformations of parallel behaviours. We assume that the parallel behaviours and their transformed versions are both available as KPN models. The KPN to ADDG transformation scheme is applied for this purpose. Specifically, we model both the input and the transformed KPNs as ADDGs and apply our ADDG based equivalence checking method for establishing the

equivalence between the two KPNs. We then show that the commonly used parallel transformations techniques can be verified in our verification framework. Experimental results are presented.

1.5 Organization of the thesis

The rest of the thesis is organized in the following manner.

Chapter 2 provides a detailed literature survey on the applications of commonly used behavioural transformations and their existing verification methods. In the process, it identifies the limitations of the state of the art verification methods and underlines the objectives of the thesis.

Chapter 3 discusses a path extension based equivalence checking of code motion transformations. The issues addressed in the chapter is illustrated first. The FSMD model is then introduced. A normalization procedure over integer arithmetic is given next. The notion of path extension and its enhancement for verifying non-uniform code motions are described next. The verification method is then given. The chapter also provides a detailed theoretical analysis of the method and some experimental results with the implementation.

Chapter 4 discusses the high-level to RTL translation verification method. The basic construction mechanism of the FSMDs from the RTL designs is discussed first. How the basic method is enhanced to handle multicyle, pipelined and chained operations is given next. The overall construction mechanism is then discussed. Termination, soundness and completeness of the method have been proved and the complexity of the method analyzed. The verification of the data-path and the controller during FSMD construction is given next. The chapter then discusses the applicability of this method to verify RTL level transformations and finally, provides some experimental results on this method.

Chapter 5 discusses the verification of loop and arithmetic transformations of array intensive behaviours. The ADDG model and its different entities are formally defined and elaborated with examples first. The method to obtain an ADDG from a sequential behaviour is then presented. A normalization method and some simplification rules of normalized expressions for array intensive arithmetic expressions are introduced. The chapter defines the notion of slice based equivalence of ADDGs next. The correctness and complexity of the method have been analyzed subsequently. The chapter finally provides some experimental results and error diagnoses of the presented method.

Chapter 6 discusses the verification of sequential to parallel and parallel to parallel code transformations. The objectives of the work are illustrated first. The verification framework is given next. The chapter then provides a mechanism of modelling a KPN as an ADDG. The proof of correctness of this translation is given next. The chapter then discusses the commonly applied KPN level transformations and shows that our verification framework is strong enough to verify those transformations. Finally, some experimental results are provided.

Chapter 7 concludes our study in the domain of verification of behaviour transformations during embedded system design and discusses potential future research directions in this field.