NORTHWESTERN UNIVERSITY

System–Level Synthesis and Verification

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Nikolaos Liveris

EVANSTON, ILLINOIS

December 2008

ABSTRACT

System–Level Synthesis and Verification

Nikolaos Liveris

As device sizes decrease, more functionality can be placed in an integrated circuit. Therefore, the design complexity of these circuits increases. To deal with complexity, designers move to higher abstraction levels. Currently, the highest abstraction level is the system-level. In our work we investigate the synthesis and verification problem at the system-level.

We examine ways to increase the energy efficiency of specific system-level designs. Moreover, we propose an algorithm to retime a system-level description, so that its performance becomes optimal. Retiming is a powerful synthesis operation that can be used to change the schedule of a design. We investigate the optimization power of synthesis operations, like retiming, and propose a sequence of synthesis operations that is complete for the transformation of sequential circuits.

The verification problem is hard. Checking equivalence between two designs or checking whether a design satisfies a given assertion is proven to have high computational complexity in the general case. We describe ways to simplify the verification problem. First, we show that the verification problem can be simplified by considering it during

synthesis without restricting the optimization power of the synthesis operations. Then we show how abstraction can enable the use of efficient automated verification tools.

Acknowledgements

I would like to express my gratitude to my advisors Professor Hai Zhou and Dr. Prith Banerjee. This work would not have been possible without their support and guidance. Dr. Banerjee brought me into the field of VLSI design automation and continued advising me even after he left Northwestern University. Professor Zhou taught me the art of algorithm design and the importance of independent thinking. His deep knowledge and understanding of the field have made this work achievable. Both my advisors have been a constant source of encouragement and moral support.

I would like to thank the members of my dissertation committee, Professors Robert Dick and Seda Memik, for their interest in and suggestions on my work.

I would also like to thank the members of the NuCAD research group for all the interesting discussions we had the last years, Dr. Ruiming Chen, Debasish Das, Bach Ha, Dr. Chuan Lin, Dr. Debjit Sinha, Stephen Tarzia, and Dr. Jia Wang.

Part of my research was conducted during my internships at Calypto Design Systems. I am grateful to Dr. Deepak Goyal, Dr. Gagan Hasteer, Dr. Sumit Roy, Nikhil Sharma, and Anup Sultania for the discussions and useful feedback on my research.

This is an opportunity to thank my advisor at the University of Patras, Professor Costas Goutis and my advisor at IMEC, Professor Francky Catthoor for teaching me how to conduct independent research.

I am also grateful to my family for their support and their love. I especially thank my parents, Dimitris and Vivi, for the sacrifices they have made for me. Finally, I am highly indebted to my wife, Elena, for all her patience, support, and love throughout these years.

NIKOLAOS LIVERIS

EVANSTON, SEPTEMBER 2008

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Over the last years transistor sizes have dramatically shrunk. This reduction enabled designers to put more transistors and because of that more functionality on a single chip. As a result, the complexity of the designs has increased. To deal with the increasing complexity designers use abstraction, i.e., only the necessary details of the design are considered before a specific design decision is made. All other information is abstracted.

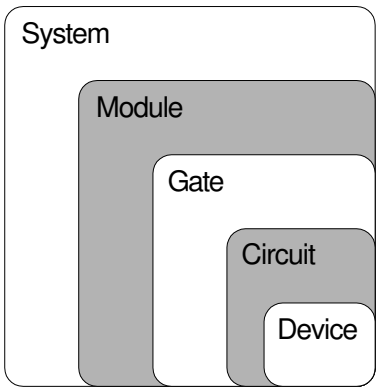


Figure 1.1. The different abstraction levels for a digital integrated circuit.

In Figure 1.1 the different abstraction levels that are used for digital integrated circuits can be seen [75]. The lowest level is the device level at which the physical behavior of the semiconductor devices is considered. Next is the circuit level at which the design is represented as a set of interconnected transistors. At the gate level the building blocks are gates and the transistor-level implementation of each gate is abstracted. At the next abstraction level the basic building block is the module, e.g. an adder, or a multiplier.

Finally, the highest abstraction level is the system level, at which the building blocks are systems, e.g., processors, RAMs, or IP cores.

At each abstraction level design decisions are made that constrain the design space. This process is called synthesis. Another way to describe synthesis is as the process that produces a structural representation from a behavioral representation at each level of abstraction [26]. The structural representation is a more detailed view of the design and contains all the constraints added by the synthesis process. The behaviors of the constrained design are a subset of the behaviors that are allowed by the initial description. Therefore, the constrained design is called an implementation and the initial description a specification at each abstraction level [54].

After synthesis the designer needs to certify that the result of synthesis is correct. This process is called verification and in most cases it is much harder than synthesis. Verification checks whether certain assertions are valid for the design, whether the constrained design is equivalent to the initial design, or whether each behavior of the implementation is a valid behavior of the specification. Abstraction is also critical for the verification task, as we will see in subsequent chapters.

The characteristics of system-level descriptions vary based on the application domain, the design restrictions, and the designer's style. The system-level design space is called in some cases a continuum [17]. In Figure 1.2 a graphical representation of the continuum can be seen. The horizontal axis represents the abstraction level used for the data types of the application, while the vertical axis represents the sequential abstraction.

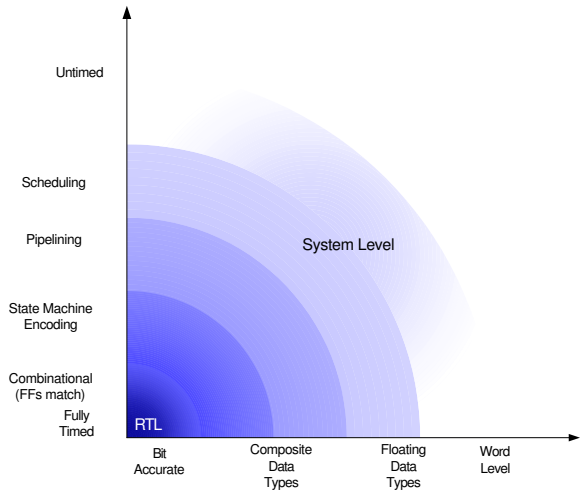


Figure 1.2. The System-Level continuum adopted from [17].

In the next two sections we introduce system-level synthesis (Section 1.1) and verification (Section 1.2). In Section 1.3 we describe the related work in this field. Then in Section 1.4 we give a roadmap of this thesis.

1.1. Synthesis at the System-Level

The highest-level of abstraction for digital circuits is the system-level. At this level decisions are made for the execution of the processes of the system. Such decisions could specify, for example, the time a process should be executed and the kind of resources that are needed for the process's execution. After those design decisions are made, the design space is more constrained and the designer can move to a lower level of abstraction. When done automatically, this task of constraining the design by making specific decisions is called synthesis.

In Figure 1.3 a possible flow for system-level synthesis can be seen. The input is the specification of the system in a high-level language, such as C,C++ [80], SystemC [71],

and some design requirements. The design requirements are associated with some of the quality metrics of the design, e.g., performance, energy consumption, area.

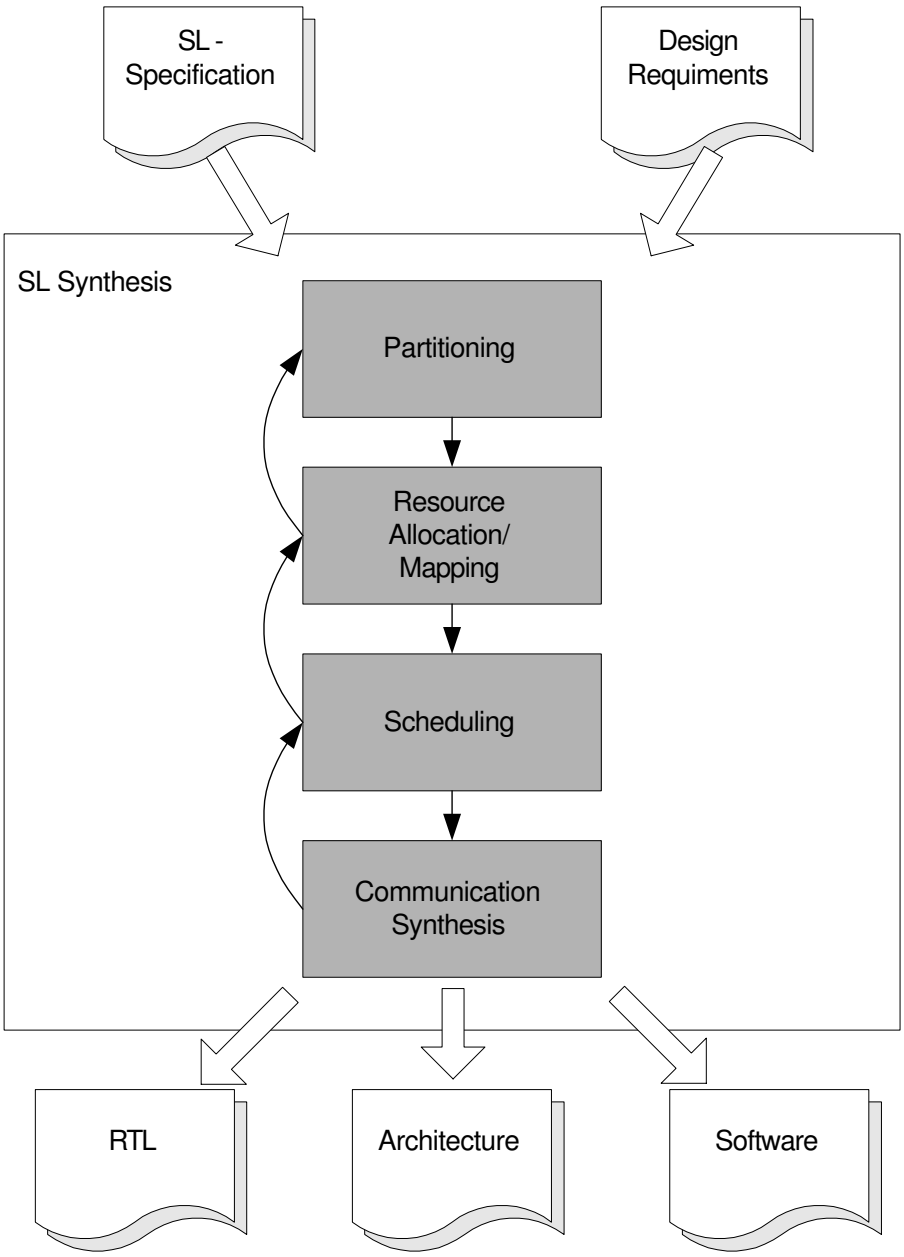


Figure 1.3. System-Level synthesis.

Typical tasks that are executed during system-level synthesis are partitioning [29, 53], resource allocation and mapping [82, 46], scheduling, and communication synthesis (Chapters 2–4). First the specification is partitioned into processes with different characteristics. The reason is that different synthesis approaches are used for processes with different characteristics. For example, processes that are dominated by arithmetic operations can be implemented in hardware, while processes with data-dependent behavior, which cannot be predicted at compile-time, will be implemented in software. The second step is to allocate computing and memory resources for these processes. Some of the processes are going to be mapped to specific IP cores. During the next step, i.e., scheduling, the starting time of each process is determined. During the communication synthesis the way the processes communicate is defined.

The four system-level synthesis tasks are related and the results of one task affect the design space of the others. The order in which these tasks are executed could vary depending on the application and the requirements. Figure 1.3 shows one possible order.

The output of system-level synthesis is an RTL (Register Transfer Level) [26] description for the processes that are going to be implemented in hardware. Software is generated for the processes that are going to be executed on programmable cores. Moreover, a description of the architecture is included in the output. The description includes the processors or other resources that are needed and the communication architecture. The communication architecture can be the definition of a bus architecture or a set of point-to-point connections or a network topology.

In our work we focus on scheduling, communication synthesis, and RTL generation. The synthesis operations that we use are applicable to the structural representation of the system-level description.

1.2. Verification at the System-Level

The result of synthesis is to produce another more constrained design whose behaviors are a subset of the behaviors that the original design could have. It is important to be able to certify that the result of synthesis is correct. This process is called verification and is harder than synthesis.

For descriptions at different points of the system-level continuum (Figure 1.2) different abstractions and verification algorithms can be used. Moreover, the verification approach is different if equivalence between two designs or the validity of an assertion on a design is checked.

The input to equivalence checking is two designs, one is the specification or golden model and the second is the implementation or the result of synthesis. The purpose of equivalence checking is to prove that the two designs have the same behavior for any possible input or to provide a counterexample that shows the opposite. At the RTL, if the registers of the two designs can be matched, combinational techniques can be used. However, at higher abstraction levels at which we are not provided with a relation between the state variables of the two designs, the verification approach is more complicated. The general problem is PSPACE complete, as we will see in subsequent chapters.

The input to assertion checking is a design and a property to be checked. The property could be as specific as a boolean condition over the value of a variable or could be as

general as the temporal specification of a design. The verification algorithm checks that all possible behaviors allowed by the design and its environment do not violate the assertion.

1.3. Related Work

Our focus for system-level synthesis will be on the scheduling and communication synthesis steps. As input to the algorithms of those steps we will consider behavioral and structural models that capture the relevant details of the system-level description and any constraints imposed by the previous steps of partitioning and resource allocation. The model we will use most frequently is the Synchronous Data Flow.

Synchronous Dataflow Graphs (SDFs) are considered a useful way to model Digital Signal Processing applications [55]. This is because in most cases the portions of DSP applications, where most of the execution-time is spent, can be described by processes or actors with constant rates of data consumption and production. Moreover, efficient memory and execution-time minimization algorithms have been developed for SDF graphs [12, 35].

Energy consumption is one quality metric for digital integrated circuits. The main sources of energy consumption are dynamic and static power dissipation. Static or leakage power is expected to become the dominant power dissipation component for future technologies [37]. Therefore, techniques to reduce the leakage power are needed.

Work on leakage reduction at the higher levels of design has been focused on replacing cells or submodules of the design with ones with the same functionality but higher threshold voltage (e.g. [47]). Although these techniques can lead to significant reductions, they are not applicable to parts of the design that come as hard cores or when the available time slack changes, even with a low frequency, e.g. by the user of the system. In these

cases, techniques are needed that are adaptive to environment changes and do not require resynthesis of IP cores. Such techniques include Dynamic Voltage Scaling [15], Adaptive Body Biasing [48], and Power Gating [37]. In Chapter 2 we will present a technique that uses power gating to reduce energy consumption.

Another quality metric of digital designs is performance. Most applications come with tight throughput or latency constraints. Those constraints are dealt with during the scheduling task. During that task synthesis operations are used to improve the performance of the design. One of those operations, retiming, has been used widely in the past to optimize the cycle time or resources of gate-level graph representations [57, 60, 88]. A lot of work has been done on extending retiming on SDF graphs [70, 91]. More specifically, retiming has been proposed to facilitate vectorization [90] and to minimize the cycle length of these graphs [70].

Govindarajan and Gao have proposed an algorithm to determine a non-blocking schedule for an SDF graph for maximum throughput [35]. However, there are design cases in which a non-blocking schedule is not feasible. This happens when a part of the application's behavior is determined dynamically at run-time, or when some of the application's processes share resources with higher-priority processes. These processes are normally executed on a programmable processor, while the computationally expensive part of the application is run on dedicated resources, has predictable execution time, and is conveniently modeled as an SDF. In case there are data dependencies between the SDF actors and the processes executed on the programmable processors, a non-blocking schedule may not be feasible. Then the blocking schedule with the minimum cycle length is equivalent to minimum latency of the static part of the application (Chapter 3).

O'Neil and Sha proposed an algorithm to reduce the clock cycle of a graph below a threshold using retiming [70]. In Chapter 3 we propose an optimal algorithm for retiming SDF graphs. The purpose is to minimize the length of the complete cycle of a SDF graph. The algorithm produces better results than any existing approach and it is orders of magnitude faster than O'Neil's algorithm.

Retiming can be used in conjunction with another synthesis operation called resynthesis. Resynthesis is an operation that changes the circuit structure without changing the function of the combinational logic. There is a long history of investigations and debates on whether a sequence of retiming and resynthesis is complete for any sequentially equivalent transformation.

Malik [62] gave the first (partial) positive answer to this question. He proved that retiming and resynthesis are complete for any state re-encoding, and for some other transformations. Zhou et al. [89] provided the first negative answer by proving that some sequentially equivalent transformations cannot be done by retiming and resynthesis, which also helped to discover and fix an error in Malik's result [76]. The sweep operation, which adds or removes registers not used by any output, is needed for these transformations. However, it is an open question whether retiming and resynthesis with sweep are complete for general sequential transformations. In Chapter 4, we provide a complete answer to the open question.

Zhou et al. [89] also started an investigation on the complexity of retiming and resynthesis verification problem. Since the general sequential equivalence verification is PSPACE-complete, a different complexity category may indicate that the gap between retiming and resynthesis and sequential transformation is big. Jiang and Brayton [39]

later showed that the complexity of retiming and resynthesis verification is also PSPACE-complete. We examine their proof and point out parts that are unclear. Based on those we consider the membership of retiming and resynthesis verification an open question.

Despite its optimization power, the Retiming and Resynthesis (RnR) sequence is not widely used due to the complexity of checking sequential equivalence [39] between the initial and final design. There is a need, therefore, for efficient verification methods that preserve the optimization power of the retiming and resynthesis sequence.

Van Eijk developed an efficient method for checking sequential equivalence between two designs that is based on finding equivalent signals in the two circuits [85]. Jiang et. al. showed that the method is complete for sequences of retiming and resynthesis transformations with no more than one resynthesis step [40]. If more than one resynthesis step is applied and the verification procedure shows that the outputs are not equivalent, no conclusion can be drawn.

Ashar et. al. demonstrated that circuits with the Complete-1-Distinguishability (C-1-D) property can be verified with an efficient and complete method [4]. In C-1-D circuits each pair of distinct states produces different output values for some input and, therefore, each state is distinguishable from any other in a single cycle. If one of the two circuits to be checked for equivalence satisfies the C-1-D property, sequential equivalence checking can be reduced to combinational equivalence checking. Not all circuits satisfy C-1-D and, therefore, the authors developed a method to enforce this property by modifying the structure of the circuit. However, a side effect of the modifications to enforce C-1-D is that the optimization power of retiming and resynthesis is reduced.

A complete method to check for sequential equivalence of two circuits without restrictions on the synthesis part is model checking, i.e., reachability analysis [23]. Starting with the initial state of the circuits a forward traversal of the state space can be performed to check whether a "bad state", i.e., a state that shows the two circuits are not equivalent, can be reached. During each iteration the method uses the next state relation to increase the set of reachable states. In backward reachability analysis the process starts from the set of bad states and checks whether an initial state is reachable using the inverse of the next state relation. The number of iterations that this method requires to produce a useful answer is generally hard to compute. Without this bound, if the set of reachable states does not converge after a specific number of iterations, no conclusion can be drawn for the correctness of the transformations.

To improve the efficiency of reachability analysis and reduce the number of iterations without destroying completeness, a number of structural optimizations have been proposed [50, 36]. For example, retiming can be used to reduce the number of variables before the traversal starts. These techniques can be used in conjunction with the ideas proposed in Chapter 5.

The approach we describe in that Chapter targets the equivalence checking of a pair of circuits, one of which has been obtained from the other by a sequence of retiming and resynthesis transformations. We extend the C-1-D property to C-k-D, where $k \in \mathbb{N}$. A circuit fulfills the C-k-D property if every two non-equivalent states can be distinguished in k cycles or less.

In addition to model checking, theorem provers can be used for formal verification. For theorem provers both the system and its desired properties are expressed as formulas

in some mathematical logic and the theorem prover finds a proof from axioms of the system. SVC [8] and its enhanced version CVC [81] are automatic theorem provers for first order logic. PVS [72] combines decision procedures and model checking with interactive proof. Theorem provers in contrast to model checkers can handle infinite state spaces but generally require manual intervention and are hard to use.

CBMC [19] uses a system-level specification of the circuit, written in C, to verify the RTL model. The techniques to capture the model are the same as in BMC approaches and a bit-level SAT solver [67] is used to produce a counterexample or to prove the correctness of the assertions.

In Chapter 6 we describe an alternative approach to CBMC for verifying properties of an RTL description using its system level specification. The approach is orders of magnitude faster than CBMC for computational intensive applications by sacrificing bit-level accuracy, which may not be needed during the early stages of the verification process. The back-end tool used in the framework is Mathematica, a well known commercial symbolic analysis tool.

Even though using abstraction for the data types can speed up verification for some designs, in general the large number of cores in a System-on-Chip can make verification intractable. Because the number of cores is large, the algorithms developed for the communication of those cores are parameterized, i.e., they work for any number of cores. These algorithms are distributed and are executed by all cores to ensure cooperation when resources are shared. In such cases, the SoC can be thought as a distributed system.

Automated methods for the verification of distributed systems can only be applied to relatively small finite-state systems. However, most distributed algorithms are specified

for an arbitrary number of processes. More specifically, the number N of processes present in the distributed system is a parameter and the algorithm is expected to work for any valid value of the parameter. We call these systems parameterized systems. An instance of the parameterized system is the system built for a specific value of N. Although automated methods, i.e., model checking, can be used for the verification of instances with small number of processes, they can neither be efficiently applied to large instances nor prove that all possible instances of a system are correct. In those cases abstraction is necessary.

Using abstraction a finite-state system can be derived from a parameterized system. We call the derived system the abstract system. If the correctness condition holds for the abstract system, then it holds for all instances of the parameterized system. Since the state space of the abstract system is finite, model-checking can be used to check whether it satisfies the correctness condition.

A number of abstraction methods have been developed for high-atomicity parameterized systems [21, 74, 9, 30]. High-atomicity parameterized systems are systems in which the number of variables each process can read or write in one atomic step increases, as the parameter N increases. Since for large N such communication operations become very expensive, we focus on low-atomicity systems.

Our work targets a specific class of fault-tolerant systems; self-stabilizing systems. Self-stabilizing systems are systems that automatically recover after any transient fault [27]. For those systems liveness properties, i.e., properties that specify that something good will eventually happen, are more relevant than safety properties, i.e., properties that specify that nothing bad will happen. This is because transient faults can bring the system in

any arbitrary state, making all states in the state space reachable (Section 7.2). Since we focus on self-stabilizing systems, we consider only abstraction methods for the verification of liveness properties.

For the verification of liveness properties in low-atomicity parameterized distributed systems two abstraction techniques have been developed: the method of invisible ranking [33] and the method of control abstraction [52, 43]. The idea behind the method of invisible ranking is to bound the number of processes needed to prove a correctness property for a class of parameterized systems [33]. The approach can be used for the verification of properties of the form $\Box(p \rightarrow \Diamond r)$, i.e., for every state satisfying assertion p there is a future state satisfying assertion r. It is not known how other liveness properties can be checked using this method. Moreover, in some cases the number of required processes is large (128 for the dining philosophers problem).

An alternative approach is the method of control abstraction. The idea behind control abstraction is to abstract away an arbitrary number of symmetric processes by using a particular process called network invariant. Then the correctness property is checked in the abstract system, which is composed of a small finite number of processes and the network invariant [43]. There are two difficulties that have restricted the applicability of this method. The first is that there is no automated method for the construction of the network invariant. Existing automated approaches for the construction of network invariants target only safety properties [58]. The second is that the network invariant must have the same set of observable variables as the system of symmetric processes abstracted by it. Because of this constraint, the usage of control abstraction has been restricted to ring topologies of processes [45], in which each process reads the variables of

only two neighbors. It has also been successfully applied on systems where the number of shared variables does not increase with the number of processes. An example is a mutual exclusion algorithm with all processes sharing only one semaphore [43].

In Chapter 7 we present an abstraction technique that builds on the theory of control abstraction. This is the first abstraction technique that can be used to prove the correctness of low-atomicity, parameterized self-stabilizing systems, whose number of observable variables may increase with the number of processes in the system. The case studies demonstrate that our abstraction technique is not trivial and can be applied to distributed algorithms to which no other abstraction technique has been successfully applied.

The derived abstract system is relatively small and its state space does not increase exponentially with the number of states of the abstracted symmetric processes, as it is the case in [74]. The proposed abstraction technique handles both weak and strong fairness constraints for the abstracted processes, as opposed to previous works [9]. Finally, because it uses syntax manipulation, the complexity of the algorithms building the transition relation is low compared to approaches that use decision procedures (MONA) [74, 9].

1.4. Roadmap of the Thesis

The goal of system-level synthesis is to constrain the behavior of the design, so that its quality metrics are optimized. There exist several quality metrics for digital circuits that depend on the intended usage of the system. In our work synthesis targets energy consumption and performance as quality metrics.

We present an algorithm for reducing the energy consumption of a system-level design. The algorithm can be applied to pipelined designs and is used to define a schedule for each pipeline stage. The energy consumption is reduced by increasing the number of consecutive cycles each hardware unit is disconnected from the power line. The number of switches from sleep to active mode for those units is also reduced. The approach is presented in Chapter 2.

A new algorithm for improving the performance of a system-level design is described in Chapter 3. The algorithm can be applied to systems with processes that have constant production and consumption rate. This type of systems is very common in the signal processing and communication domains. Our algorithm produces a static schedule with the shortest running-time and is faster than any existing approach. The algorithm uses a well-known synthesis operation called retiming [57] to improve the scheduling.

Most synthesis approaches are centered around specific synthesis operations, e.g., retiming. It is, therefore, interesting to investigate the power of those operations. In Chapter 4 we examine the power of several synthesis operations and we prove that a sequence of five operations is complete for any sequential transformation.

Even though system-level synthesis approaches can significantly improve the quality metrics of a design, their adoption is hindered by the verification complexity. Designers hesitate to use synthesis algorithms, unless they can verify their results. However, verification of sequential circuits is a hard problem. To simplify the problem we can modify synthesis operations to provide hints to the verification procedure. Moreover, for some application domains abstraction can enable the use of fast verification tools.

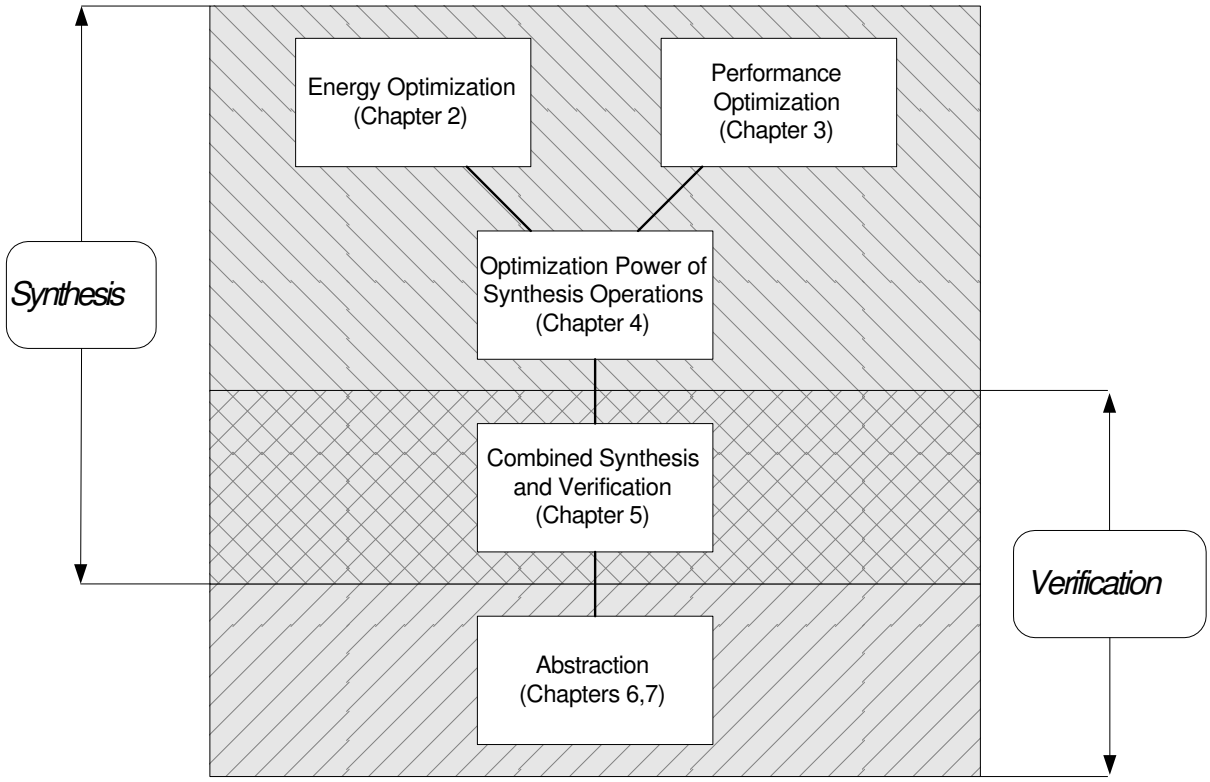


Figure 1.4. The structure of this thesis.

In Chapter 5 we consider the combined synthesis and verification problem for sequential circuits. Without reducing the optimization power or complexity of synthesis, we can simplify the verification procedure by enforcing specific properties on the design before synthesis starts. Then equivalence checking of the original and the transformed design is much faster in some cases.

In Chapter 6 we describe a new framework for verifying computation intensive applications. Computation intensive applications use a lot of arithmetic operations which are not efficiently handled by bit-level verification procedures. We sacrifice bit-level accuracy,

which may not be important at the system-level, to improve the running-time of verification. The proposed framework is based on Mathematica [87], a well-known commercial symbolic analysis tool.

In Chapter 7 we present a number of abstraction techniques that can enable the use of model checking for parameterized systems. Parameterized systems have an arbitrary number of processes with the same behavior. The abstraction techniques target self-stabilizing systems, i.e., systems that recover from any transient fault. Conditions of completeness for the abstraction technique are also defined and its effectiveness is demonstrated on a number of case studies.

Finally, in Section 8 we give our conclusions. In Figure 1.4 a flow chart of the work presented in this thesis can be seen.

CHAPTER 2

Energy Optimization of Pipelined System-Level Steaming Applications

2.1. Introduction

Synchronous Dataflow Graphs (SDFs) are considered a useful way to model Digital Signal Processing applications [55]. This is because in most cases the portions of DSP applications, where most of the execution-time is spent, can be described by processes or actors with constant rates of data consumption and production.

Energy consumption is one quality metric for digital integrated circuits. The main sources of energy consumption are dynamic and static power dissipation. Static or leakage power is expected to become the dominant power dissipation component for future technologies [37]. Therefore, techniques to reduce the leakage power are needed.

Work on leakage reduction at the higher levels of design has been focused on replacing cells or submodules of the design with ones with the same functionality but higher threshold voltage (e.g. [47]). Although these techniques can lead to significant reductions, they are not applicable to parts of the design that come as hard cores or when the available time slack changes, even with a low frequency, e.g. by the user of the system. In these cases, techniques are needed that are adaptive to environment changes and do not require resynthesis of IP cores. Such techniques include Dynamic Voltage Scaling [15], Adaptive Body Biasing [48], and Power Gating [37]. We focus on the latter technique.

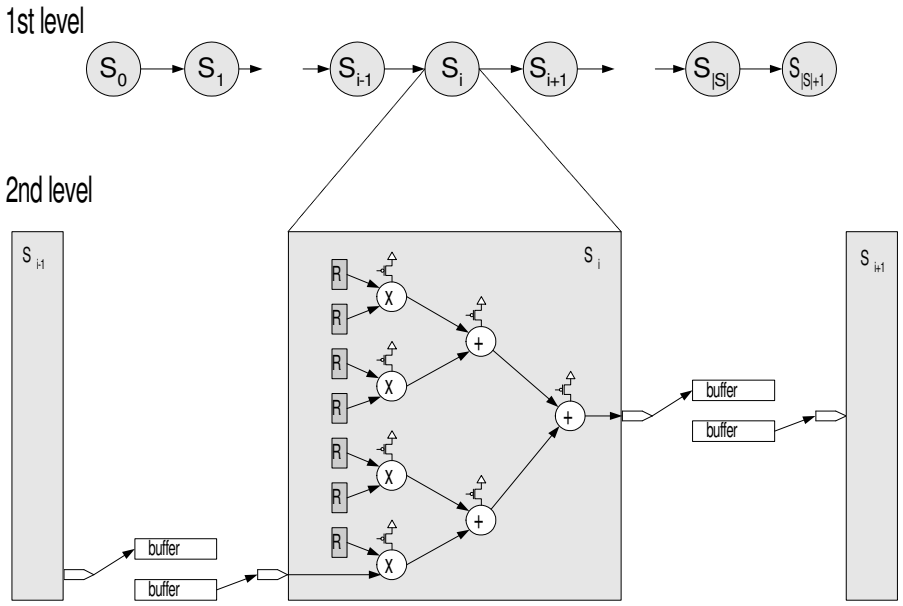


Figure 2.1. System structure. At the first level of hierarchy the system is a pipelined chain-structured graph. The processes (nodes) of the second level can be independently power gated. Cross edges between pipeline stages are implemented using buffers.

With power gating, a hardware module is shut down when it is idle. This way the stand-by leakage of the module is reduced. The switching from active to sleep mode and back to active has an energy penalty caused mainly by the loading of the nodes to normal V_{dd} levels [37]. In this work we try to decrease energy consumption by reducing the number of times the mode switch occurs.

Our approach is to try to find the number of consecutive iterations for each pipeline stage of a chain-structured SDF graph. This problem is similar to vectorization [77], but in our case instead of trying to maximize the consecutive number of executions, we try to maximize the energy savings taking into account the energy penalty paid by adding more buffers to each channel. Dynamic programming techniques have been used to determine a

schedule for a chain-structured SDF, so that the memory requirements are minimized [68]. In our approach, the buffer requirements are increased, whenever this increase leads to a reduction of the total energy consumption.

The throughput of the application does not change after applying our method. Moreover, our method guarantees that any latency increase does not cause data loss. In general for streaming multimedia applications throughput constraints are important and less emphasis is put on latency [38]. Our technique is applicable only to streaming applications, for which a latency increase is acceptable.

In Section 2.2 we explain the model we use to describe pipelined system-level applications. Section 2.3 defines the problem we try to tackle. In Section 2.4 the theoretical issues of the problem are addressed, while Section 2.5 describes an algorithm that can be used to solve it. Finally, in Sections 2.6 and 2.7 we present experimental results and draw conclusions.

2.2. Model Description

In this section we describe the model we use for system-level pipelined applications. Table 2.1 summarizes the definitions of the symbols used in this chapter. In Figure 2.1 the structure of the model can be seen.

2.2.1. Chain-Structured SDFs

In an SDF $G = (V, E)$ each node represents a process and each edge a channel, in which the tail produces data and the head consumes data. We assume a global clock for the whole system. Functions $p : E \rightarrow \mathbb{N}$, $c : E \rightarrow \mathbb{N}$, and $w : E \rightarrow \mathbb{Z}_0^+$ represent the production, consumption rates, and the number of initial tokens (delays) of each channel.

Symbol	Definition
q_s	number of executions (instances) of stage s in one complete cycle
$p(i, i+1)$	number of tokens produced on cross edge (i,i+1) as a result of one execution of stage s_i
$c(i, i+1)$	number of tokens of cross edge (i,i+1) consumed as a result of one execution of stage s_{i+1}
$w(i, i+1)$	number of initial tokens (delays) on cross edge (i,i+1)
$b(i, i+1)$	number of buffers on cross edge (i,i+1)
$l(G_s)$	execution time in cycles for each instance of a pipeline stage s
$l(v)$	the number of cycles process $v \in V_s$ must remain active during the execution of one instance of s
$E_{sm}(v)$	energy overhead for switching modes from active to sleep and back to active for process v
$\Delta\mathrm{P}(v)$	power difference between active and sleep mode when process v is idle
L_{cc}	period of execution for a complete cycle of the pipeline (chain-structured SDF graph)
L_s	period of invocation for pipeline stage s, initially equal to $\frac{L_{cc}}{q_s}$
x_s	number of consecutive instance executions of pipeline stage s, initially equal to 1
ρ	quality metric of the solution, applicable only to unirate SDF graphs
$E_s(v)$	energy savings from a process v
$E_p(i, i+1)$	energy penalty on cross edge (i,i+1)
$E_t(\tilde{x})$	the total energy savings after subtracting the total energy penalty on the channels for a solution $\tilde{x}$
$\mathbb{N}$	the set of natural numbers
$\mathbb{Z}_0^+$	the set of non-negative integers ($\mathbb{N} \cup \{0\}$)

Table 2.1. Definition of the symbols used in this chapter.

In order for an SDF to be executable with bounded memory, the system $\Gamma\tilde{q} = 0$ should have non-trivial solutions, where Γ is the topology matrix of G [55]. The vector with the minimum positive integers in the solution space, $\tilde{q}$, is called the repetition vector and each entry represents the number of times the corresponding node should be executed during each complete cycle of the graph. An SDF is called consistent if it has a repetition vector and the system does not deadlock [73].

A proper subset of E in the graph may not have either a tail or a head. These are the input and output edges with which the SDF communicates with its environment.

In case all production and consumption rates are equal with 1, the graph is called a unirate SDF. Otherwise, it is called a multirate SDF. A unirate SDF has a repetition vector with all entries 1.

The subset of SDFs we are interested in can be represented in the first level of hierarchy as a chain-structured directed multi-graph $G = (S, E)$ [68] with nodes that are all executed in parallel. We define G as a graph with $|S|$ nodes, for which there are labels $s_1, s_2, ..., s_{|S|}$, such that each edge $e \in E$ can be directed only from s_i to s_{i+1} for any i. Therefore, there can be multiple edges between two nodes, but edges can only connect nodes, whose labels differ by one, in the direction from the smallest label to the greatest. We call these nodes *pipeline stages* or *stages* and we call the edges between pipeline stages *cross edges*.

Properties of hierarchical clustering of SDFs are described in [73]. In our case we assume the clustering has been done to satisfy an average throughput constraint for the graph and to minimize the cost of pipelining on cross edges. Here we assume that the data a stage consumes have to be available until the end of the stage's execution. Moreover, the memory to store the data produced by a stage should be available before the starting time of that stage.

All input edges of the application SDF become cross edges, whose head is s_1 and whose tail is stage s_0, which is external and we have no control over it. An external stage $s_{|S|+1}$ is defined for the output edges, as well. Each stage s is already synthesized and has an execution time of $l(s)$ cycles.

2.2.2. Processes

Each pipeline stage s can be represented by a directed graph $G_s = (V_s, E_s)$, where V_s is the set of processes and E_s is the set of edges (channels) between the processes.

Function $l : V \rightarrow \mathbb{N}$ returns the number of clock cycles process $v \in V_s$ must remain active during the execution of s.

We assume that when a process v is idle, it can be in an active and power-hungry or a sleep and power-efficient mode. The power difference is $\Delta\mathrm{P}(v) = P_{acm} - P_{slm}$, where P_{acm} and P_{slm} are the power in active and sleep mode. $P_{ac2slm}(v)$ and $P_{sl2acm}(v)$ are the average power consumptions during switching modes and $t_{ac2slm}(v)$, $t_{sl2acm}(v)$ the time periods needed for the switching. Then if v does not switch mode the total energy dissipated in the slack time is:

$$E_{ac} = \Delta\mathrm{t} \cdot P_{acm}$$

while if it is switched to sleep mode the total energy dissipated is:

$$E_{sl} = (\Delta\mathrm{t} - t_{ac2slm} - t_{sl2acm}) \cdot P_{slm} + t_{ac2slm} \cdot P_{ac2slm} + t_{sl2acm} \cdot P_{sl2acm}$$

The energy savings for switching a node v from active to sleep mode during some time interval $\Delta\mathrm{t}$, in which the process is idle, are

$$\begin{aligned} E_s(v) &= E_{ac}(v) - E_{sl}(v) \\ &= \Delta\mathrm{t} \cdot (P_{acm}(v) - P_{slm}(v)) \\ &\quad -P_{ac2slm}(v) \cdot t_{ac2slm}(v) - P_{sl2acm}(v) \cdot t_{sl2acm}(v) \\ &\quad +P_{slm}(v) \cdot t_{ac2slm}(v) + P_{slm}(v) \cdot t_{sl2acm}(v) \\ &= \Delta\mathrm{t} \cdot \Delta\mathrm{P}(v) - E_{sm}(v) \end{aligned} \tag{2.1}$$

where E_{sm}, the energy penalty paid each time node v switches mode, is

$$\begin{aligned} E_{sm}(v) &= P_{ac2slm}(v) \cdot t_{ac2slm}(v) + P_{sl2acm}(v) \cdot t_{sl2acm}(v) \\ &\quad -P_{slm}(v) \cdot t_{sl2acm}(v) - P_{slm}(v) \cdot t_{ac2slm}(v) \end{aligned}$$

We assume that P_{acm},P_{slm},P_{ac2slm},P_{sl2acm},t_{ac2slm},t_{sl2acm} are given for all nodes and we can compute E_{sm} from these values.

Note that P_{ac2slm} and P_{sl2acm} account for both the dynamic and static power. Moreover, we consider E_{sm} constant, whenever Δt is large enough so that $\Delta \mathrm{t} \cdot \Delta \mathrm{P} > E_{sm}$. If any state registers are present in a process, they are not put in sleep mode, so that the state of the process can be preserved.

While each stage is defined by its ability to be executed in parallel with other stages, each process is defined by its ability to change mode independently of other processes[1].

2.2.3. Communication Channels

Communication channels are represented by directed edges, which connect processes or pipeline stages. Each edge can be implemented as a FIFO buffer. The amount of storage required for the buffer is given by the maximum number of tokens $b(e)$ at any time on the edge, which is determined by the schedule of the SDF. Since we do not modify the schedule inside a pipeline stage, we focus on the energy consumption of cross edges only.

The energy consumed on a cross edge is an increasing non-linear function of $b(e)$ and can be different for each edge, since the size of the tokens, the interconnect, and access patterns may be different.

[1]Note that at this level each process represents a hardware unit. Since the graph G_s can have cycles and because of the definition of $l(v)$, our model does not prevent resource sharing.

We use the symbol $E_p(e, b(e))$ for the static and dynamic energy consumed on the memory implementing the channel e, if e requires memory space for $b(e)$ tokens.

2.2.4. Scheduling and Throughput

A complete cycle or iteration of the graph consists of the execution of each stage s q_s times, where q_s is the corresponding entry for s in the repetition vector. We say that there are q_s invocations or instances of s in one complete cycle of G. We denote s^i the ith instance of a stage s. Since G runs for an infinite number of times, $i \in Z^+$. For completeness we include instance s^0, which is not executed. Instance s^0 is considered to be completed before any other stage starts its first instance.

Static scheduling imposes an ordering on the execution of events. A parallel schedule is a partial order on the set of the events. The partial order can be defined by a reflexive, anti-symmetric, and transitive relation R of precedence on the events. We denote as $\alpha \preceq \beta$ or $(\alpha, \beta) \in R$ the fact that event α happens before β happens. If α and β are not ordered by the relation, $(\alpha, \beta) \notin R$ and $(\beta, \alpha) \notin R$, the two events can occur in any order, even at the same time. An event can be the starting time or the ending time of the execution of a node. We can extend this relation to the execution of instances of stages. More specifically, we denote as $\alpha^i \prec \beta^j$ the fact that the ending time of instance i of stage α happens before the starting time of instance j of stage β. The relation $\prec$ is also transitive.

The edges of the graph define precedence constraints that restrict the number of available schedules that can be generated. Since all nodes (processes and stages) may carry state from one iteration to the next $\forall k \in 1..q_v : v^{k-1} \prec v^k$.

The buffer size of a channel should be large enough to store the maximum number of tokens present at that channel at any time. Suppose that $\prec$ defines a consistent and admissible schedule, then:

$$\forall (u,v) \in E, \forall i \in \mathbb{N}, \text{ let } j_{\max} \triangleq \max\{j \mid j \in \mathbb{Z}_0^+ \wedge v^j \prec u^i\}, \text{ then}$$

$$b(u,v) = \max_{\forall i}(i \cdot p(u,v) - j_{\max} \cdot c(u,v)) + w(u,v) \tag{2.2}$$

The above formula holds because during the ith instance of the producer $(i-1) \cdot p(u,v)$ tokens have already been produced and $p(u,v)$ are being produced during that iteration. Meanwhile, $j_{\max}$ instances of the consumer have already completed execution and, therefore, $j_{\max} \cdot c(u,v)$ tokens have been consumed. To the total number of tokens present we need to add the $w(u,v)$ initial tokens.

We assume that the token production at the inputs is periodic. That means that if a complete cycle is executed within L_{cc}, then for each input edge i the period is $L_i = \frac{L_{cc}}{q_i}$, where q_i is the number of instances in a complete cycle. Note that, since the input graph is assumed to be consistent, we do not need to worry about the existence of the q_i values. Each stage should have an average invocation period of $L_s = \frac{L_{cc}}{q_s}$. Therefore, $l(G_s) \leq L_s$.

2.3. Problem Formulation

As we saw in Equation 2.1, energy can be saved by switching the operation mode of some processes when enough idle time is available. One way to increase the energy savings could be to consecutively execute the stage for an integer number of times $x > 1$ and then allow its processes to be in sleep mode for a longer interval. This may increase the buffer

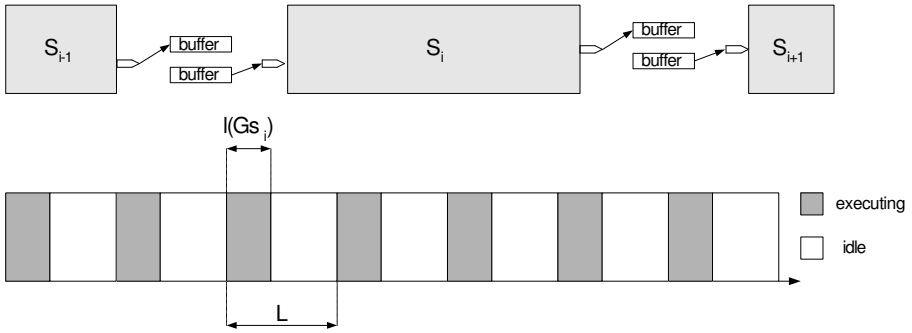


Figure 2.2. Execution of the pipeline stage s_i in the initial configuration ($x_i = 1$).

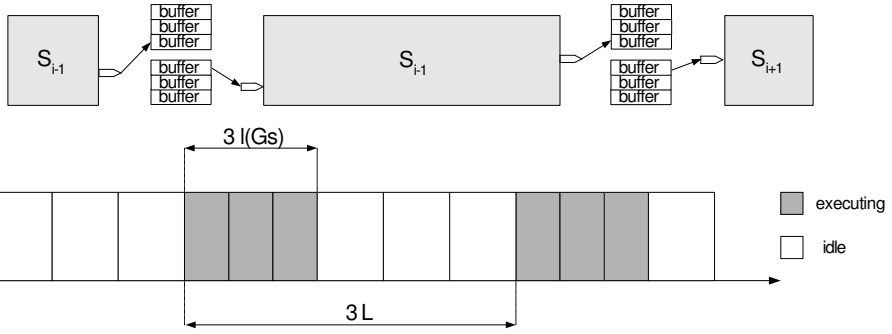


Figure 2.3. Execution of the pipeline stage s_i after the transformation ($x_i = 3$).

requirements for the input and output channels. However, the switching mode penalty can now be shared across different instances for some of the processes of the stage.

The transformation can be described as replacing stage s by s', whose firing rules can be derived by multiplying by x the number of required inputs tokens for s. Moreover, s and s' have the same process graph, which for s' is repeated x times for each invocation, and, therefore, $l(G_{s'}) = x \cdot l(G_s)$. For the edges connected to s $\forall (t, s) \in E : c(t, s') = x \cdot c(t, s)$ and $\forall (s, t) \in E : p(s', t) = x \cdot p(s, t)$.

It is easy to prove that the topology matrix of the graph G after the transformation has the same rank and since the graph is acyclic, the graph is still consistent [55].

An example is shown in Figures 2.2 and 2.3. In the first figure for stage s_i the x value equals 1. Every L cycles s_i is idle for $L - l(G_{s_i})$ cycles. If this interval is long enough, some of the processes of s_i can be switched to sleep mode. The penalty for switching from active to sleep and back to active is E_{sm} every L cycles for those processes. In Figure 2.3 the addition of 2 extra buffers allows s_i to execute for three consecutive times. The idle time increases and potentially more processes can be shut down. Besides that, the penalty for the mode switch E_{sm} is paid once every $3 \cdot L$ cycles for each process. If there is a change in the input rate and the slack becomes zero, the two additional buffers can be shut down and the stage can operate as in the first case. We assume that such changes happen with a very low frequency, e.g. the changes are caused by the user of the system, and there is a small set of predefined values for the input rate. For each of these values we solve the following problem.

Given a multirate, consistent, hierarchical graph $G = (V, E)$ with the first level of hierarchy being a chain-structured multigraph, and a quality metric ρ, find the number of consecutive executions $x_s \in \mathbb{N}$ for each stage s, so that the energy savings are not less than $(1 - \rho) \cdot E_{max}$, where E_{max} are the maximum energy savings that can theoretically be achieved by any solution to this problem.

2.4. Theoretical Exploration

In this section we reduce the search space of the solution. The solution space of the problem is $\mathbb{N}^{|S|}$. Using properties of the problem, the quality metric, and the energy penalty on the additional buffers we find an upper bound on the x values, making the

solution space finite. This upper bound affects the complexity of the proposed algorithm and its running-time as shown in Sections 2.5 and 2.6.

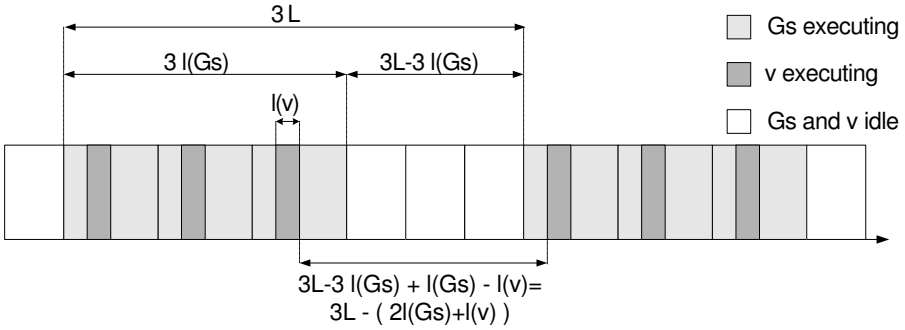


Figure 2.4. Idle time for type-1 processes.

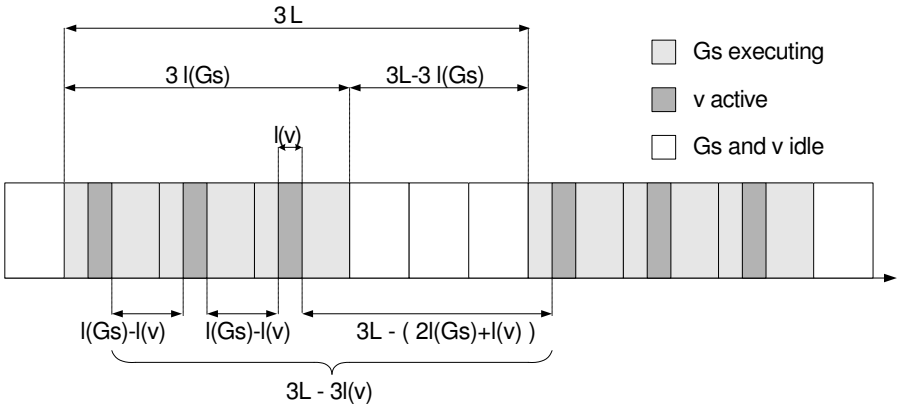


Figure 2.5. Idle time for type-2 processes.

2.4.1. Energy Savings on Processes

Using Equation (2.1) we can explore the energy savings that can be obtained by any process. Suppose that G_s is the graph representing pipeline stage s and x_s is the number of consecutive executions of G_s. We can distinguish two types of processes.

Type-1 Processes. Processes $v \in G_s$ for which

$$(l(G_s) - l(v)) \cdot \Delta \mathrm{P}(v) < E_{sm}(v) \tag{2.3}$$

are Type-1 processes.

Process v is invoked x_s times with a period of $l(G_s)$ cycles (Figure 2.4). Let $st(v)$ and $et(v)$, $st(G_s)$ and $et(G_s)$ be the start and end time of intervals $l(v)$, $l(G_s)$, respectively. In the first iteration v can be invoked after $st(v) - st(G_s)$ cycles and in the x_sth iteration it can be put to sleep mode for $et(G_s) - et(v)$. Therefore, in the first and last iterations v is in active mode for $l(G_s) + l(v)$. In the rest $x_s - 2$ iterations v is not switched to sleep mode, since Inequality (2.3) suggests that this would cause an energy loss. Therefore, the total time spent in active mode after $x_s \cdot L_s$ cycles is $(x_s - 2) \cdot l(G_s) + l(G_s) + l(v)$ or $(x_s - 1) \cdot l(G_s) + l(v)$ and the energy savings in this case are:

$$E_s(v) = (x_s \cdot L_s - (x_s - 1) \cdot l(G_s) - l(v))\Delta\mathrm{P}(v) - E_{sm}(v)$$

Therefore, on average the energy savings in L_s cycles are:

$$E_s(v) = (L_s - \frac{(x_s - 1)l(G_s) + l(v)}{x_s})\Delta\mathrm{P}(v) - \frac{E_{sm}(v)}{x_s} \tag{2.4}$$

Lemma 2.1. *The energy savings after L_s cycles of Type-1 process $v \in G_s$ are upper bounded by $(L_s - l(G_s)) \cdot \Delta\mathrm{P}(v)$.*

Proof. : Follows from Equation 2.4 for $x \to \infty$. □

From Equation (2.4) we can express the energy savings difference obtained by increasing x_s from x_1 to x_2 as

$$\Delta E_s(v)(x_2, x_1) = \frac{x_2 - x_1}{x_2 \cdot x_1}\big(E_{sm}(v) - \Delta\mathrm{P}(v) \cdot (l(G_s) - l(v))\big) \tag{2.5}$$

which is always greater than zero, since $x_2 > x_1$ and because of Inequality (2.3). Since $\Delta E_s(v)(x_2, x_1)$ is positive, $E_s(v)$ is an increasing function of x_s for all Type-1 processes.

Type-2 Processes. Processes $v \in G_s$ for which

$$(l(G_s) - l(v)) \cdot \Delta P(v) \geq E_{sm}(v) \tag{2.6}$$

are classified as Type-2 processes.

In this case during each of the $x - 1$ executions of the pipeline stage G_s, the process can be put to idle mode for $l(G_s) - l(v)$ cycles.

Lemma 2.2. *The energy savings of a Type-2 process $v \in G_s$ are independent of x_s.*

Proof. Let x_s be the number of consecutive executions of G_s. Then the energy savings for process v in $x_s \cdot L_s$ cycles are:

$$\begin{aligned} E_s(v) &= \underbrace{(x_s \cdot L_s - (x_s - 1) \cdot l(G_s) - l(v)) \cdot \Delta \mathrm{P}(v) - E_{sm}(v)}_{\text{savings due to idle time after the } x_s\text{th ex. of } G_s} + \\ &\quad + \underbrace{(x_s - 1) \cdot \big((l(G_s) - l(v)) \cdot \Delta \mathrm{P}(v) - E_{sm}(v)\big)}_{\text{savings during the first } x_s - 1 \text{ ex. of } G_s} \\ &= (x_s \cdot L_s - x_s \cdot l(v)) \cdot \Delta \mathrm{P}(v) - x_s \cdot E_{sm}(v) \end{aligned}$$

which means that the energy savings in L_s cycles are

$$E_s(v) = (L_s - l(v)) \cdot \Delta \mathrm{P}(v) - E_{sm}(v)$$

Therefore, the energy savings are independent of x_s. □

For both Type-1 and Type-2 processes we need to multiply the above findings for L_s by q_s to obtain the energy savings in L_{cc} cycles.

2.4.2. Energy Penalty on Edges

In Section 2.2.3 we saw that the energy penalty on an edge is a non-linear, increasing function $E(e, b(e))$ with respect to the buffer size $b(e)$. Therefore, it is important to study how the buffer size of a cross edge is affected by the transformation, in order to estimate the energy penalty.

Determining the minimum buffer sizes for a sequential deadlock free schedule has been done in the past [2]. However, in our case we want to find the buffer sizes for a given parallel schedule, for which we are allowed to make as few assumptions as possible. For that reason we use formula (2.2).

A simplistic approach would be to consider the buffer size of a cross edge to be an increasing function of the x values of the stages. Even though this approach is simplistic it helps us draw some useful conclusions for the more general cases.

Unirate Case. If the input graph is a unirate graph, a more realistic approach would be to consider the buffer size as the lcm function of the x values of the adjacent stages. In the graph before the transformation is applied $\tilde{q} = [11...1]$ and each stage is invoked once every L cycles. After the transformation the average rate of invocation for each instance remains the same. In $lcm(x_i, x_{i+1}) \cdot L$ cycles we know that s_i is executed $\frac{lcm(x_i,x_{i+1})}{x_i}$ times, which correspond to $lcm(x_i, x_{i+1})$ instances before the transformation. Moreover, s_{i+1} is invoked $\frac{lcm(x_i,x_{i+1})}{x_{i+1}}$ times during the $lcm(x_i, x_{i+1}) \cdot L$ cycles. Therefore, if $s_{i+1}^k \prec s_i^l$, then $\exists d_1 \in N$ such that $\forall l$

$$k = (\lceil \frac{l}{\frac{lcm(x_i,x_{i+1})}{x_i}} \rceil - d_1) \cdot \frac{lcm(x_i, x_{i+1})}{x_{i+1}}$$

Because of (2.2), $b(e) = \max_{\forall l}(l \cdot p(e) - k \cdot c(e)) + w(e)$. For $l = d_2 \cdot \frac{lcm(x_i, x_{i+1})}{x_i}$, where $d_2 \in \mathbb{N}$, $b(e)$ is maximized:

$$b(e) = d_2 \cdot \frac{lcm(x_i, x_{i+1})}{x_i} \cdot p(e) - (d_2 - d_1) \cdot \frac{lcm(x_i, x_{i+1})}{x_{i+1}} c(e) + w(e)$$

Since after the transformation $p(e) = x_i$ and $c(e) = x_{i+1}$,

$$b(e) = d_2 \cdot \frac{lcm(x_i, x_{i+1})}{\not{x}_i} \cdot \not{x}_i - (d_2 - d_1) \cdot \frac{lcm(x_i, x_{i+1})}{\not{x}_{i+1}} \cdot \not{x}_{i+1} + w(e)$$

$$\Rightarrow b(e) = d_1 \cdot lcm(x_i, x_{i+1}) + w(e)$$

In this case the buffer size and, because of that, the energy penalty are increasing functions with respect to the $lcm(x_i, x_{i+1})$.

Multirate Case. The multirate case is similar to the unirate case except that the entries in the repetition vector need to be taken into account as well. Without the transformation, stage s_i completes q_i executions and stage s_{i+1} completes q_{i+1} executions during one complete cycle. After the transformation that is not necessarily true. However, for the transformed graph we know that $lcm(x_i, x_{i+1}, q_i, q_{i+1})$ defines a period during which s_i completes $\frac{lcm(x_i, x_{i+1}, q_i, q_{i+1})}{x_i}$ and s_{i+1} $\frac{lcm(x_i, x_{i+1}, q_i, q_{i+1})}{x_{i+1}}$ executions. Solving as for the unirate case, we can find that the buffer sizes, $b(e) = d_1 \cdot lcm(x_i, x_{i+1}, q_i, q_{i+1}) + w(e)$ are an increasing function of $lcm(x_i, x_{i+1}, q_i, q_{i+1})$.

Since in all the above cases the information that we have about each edge is that they are increasing functions of $b(e)$, we can collapse all edges between two stages to one. The new function is given by: $E_p^{(i,i+1)} = \sum_{\forall \text{econnecting i to i+1}} E_p(e, b(e))$. Function $E_p^{(i,i+1)}$ is

also an increasing function with respect to the buffer sizes on all channels between stages i and $i+1$.

2.4.3. Energy Savings Limit and x_{max}

From the previous sections we can derive the formula for the total energy savings $E_t = \sum_{\forall s} \sum_{\forall v \in G_s} E_s(v) - \sum_{i=0}^{|S|} Ep^{(i,i+1)}$. This is the function we want to maximize. In this section we derive a bound on the x values of the stages to prune the search space of the problem. We start again from the simplistic case assuming that the buffer sizes are an increasing function of the x_is and move to more realistic cases.

We denote as $b(i)$ the buffer sizes of edges that connect stages i and $i+1$.

Let $v \in G_s$ be a Type-1 process and

$$C(v) \triangleq \frac{L \cdot \Delta\mathrm{P}(v) - l(G_s) \cdot \Delta\mathrm{P}(v)}{E_{sm}(v) - \Delta\mathrm{P}(v) \cdot (l(G_s) - l(v)))}$$

a constant for that node that depends only on the input graph. Let

$$C(G) \triangleq \min_{\forall v \in \text{Type-1}} C(v)$$

Then the following Lemma can be proved.

Lemma 2.3. *If G is a unirate graph and $b(i) = f(x_i, x_{i+1})$ are increasing functions with respect to both x_i and x_{i+1}, and $\tilde{x} = [x_1 x_2 ... x_{|S|}]$ is the optimal solution resulting in maximum total energy savings E_t^{max}, then for any $0 < \rho < 1$ and $x_{max} = \lceil \frac{1}{\rho \cdot C(G)} \rceil$, there exists $\tilde{x}' = [x'_1 x'_2 ... x'_{|S|}]$ with $\forall i \in 1..|S| : 1 \leq x'_i \leq x_{max}$, for which the total energy savings E'_t are greater or equal to $(1-\rho) \cdot E_t^{max}$.*

Proof. Starting with $\tilde{x} = [x_1 x_2 ... x_{|S|}]$ we can construct $\tilde{x}' = [x'_1 x'_2 ... x'_{|S|}]$ by letting:

$$x'_i = \begin{cases} x_i & : \quad x_i \leq x_{max} \\ x_{max} & : \quad x_i > x_{max} \end{cases}$$

Suppose that E_t^{max} are the total energy savings obtained by $\tilde{x}$ and E'_t are the total energy savings obtained by $\tilde{x}'$.

Energy savings for all Type-2 processes are the same for $\tilde{x}$ and $\tilde{x}'$, since they are independent of the x values (Lemma 2.2).

The values of $\tilde{b} = [f(x_0, x_1)f(x_1, x_2)...f(x_{|S|}, x_{|S|+1})]$ are all greater or equal to the values $\tilde{b}' = [f(x_0, x'_1)f(x'_1, x'_2)...f(x'_{|S|}, x_{|S|+1})]$. (The values of x_0 and $x_{|S|+1}$ cannot change as stages 0 and $|S|+1$ are external). And since $E_P^{(i,i+1)}(b(i))$ is also an increasing function of $b(i)$, the energy penalty on edges for $\tilde{x}'$ are less or equal to the ones for $\tilde{x}$.

Therefore, the total energy for $\tilde{x}'$ is at least as much as for $\tilde{x}$ considering Type-2 processes and cross edges. The energy savings difference for Type-1 processes is the upper bound of the energy difference $E_t^{max} - E_t^{new}$. For any stage s_i for which $x_i \leq x_{max}$ the energy difference is again zero. For each $v \in G_{si}$ with v being a Type-1 process and $x_i > x_{max}$, we have from Equation (2.5) that

$$E_s(v, x_i) - E_s(v, x_{max}) = \frac{x_i - x_{max}}{x_i \cdot x_{max}}(E_{sm}(v) - \Delta\mathrm{P}(v) \cdot (l(G) - l(v)))$$

for L cycles.

We want $\frac{E_s(v,x_i)-E_s(v,x_{max})}{E_s(v,x_i)} < \rho$, where ρ is a real number with $0 < \rho < 1$, which denotes the desired quality ratio, for all Type-1 processes v.

$$\frac{E_s(v,x_i) - E_s(v,x_{max})}{E_s(v,x_i)} \leq \rho \Leftrightarrow$$

$$\overset{\text{Lemma 1}}{\Leftrightarrow} \quad \frac{E_s(v,x_i) - E_s(v,x_{max})}{L \cdot \Delta\mathrm{P}(v) - l(G) \cdot \Delta\mathrm{P}(v)} \leq \rho \Leftrightarrow$$

$$\overset{\text{Equation 2.5}}{\Leftrightarrow} \quad \frac{x_i - x_{max}}{x_i \cdot x_{max}}(E_{sm}(v) - \Delta\mathrm{P}(v) \cdot (l(G) - l(v))) \leq$$

$$(L \cdot \Delta\mathrm{P}(v) - l(G) \cdot \Delta\mathrm{P}(v)) \cdot \rho \Leftrightarrow$$

$$\overset{C(v)=\frac{L\cdot\Delta\mathrm{P}(v)-l(G)\cdot\Delta\mathrm{P}(v)}{E_{sm}(v)-\Delta\mathrm{P}(v)\cdot(l(G)-l(v)))}}{\Leftrightarrow} \quad \frac{x_i - x_{max}}{x_i \cdot x_{max}} \leq \rho \cdot C(v) \Leftrightarrow$$

$$\Leftrightarrow \quad \frac{1}{\rho \cdot C(v) + \frac{1}{x_i}} \leq x_{max} \Leftrightarrow$$

$$\overset{x_{max}\in\mathbb{N}}{\Leftrightarrow} \quad \text{it is sufficient:} \lceil \frac{1}{\rho \cdot C(v)} \rceil \leq x_{max}$$

where $C(v)$ depends on the input graph. For x_{max} of the graph G we need to find $C(G) = \min_{\forall v \in \text{Type-1}} C(v)$.

For that x_{max} the total energy savings E'_t for $\tilde{x}'$ are $(1-\rho) \cdot E_t^{max} \leq E'_t$. □

For example, if the designer chooses $\rho = 0.05$, we can find x_{max} from the input graph and ρ. Then Lemma 2.3 states that there exists $\tilde{x}'$, whose entries are all less or equal to x_{max} and the energy savings for $\tilde{x}'$ are $E'_t \geq 0.95 \cdot E_t^{max}$.

Unirate Graphs. A similar approach can be followed for $b(i) = f_i(lcm(x_i, x_{i+1}))$, where for all $i \in 1..|S|$, $f_i : \mathbb{N} \to \mathbb{N}$ is an increasing function.

Lemma 2.4. *If G is a unirate graph, $f_i : \mathbb{N} \to \mathbb{N}$ is an increasing function, $b(i) = f_i(lcm(x_i, x_{i+1}))$ for each cross edge $(i, i+1)$, and $\tilde{x} = [x_1 x_2 ... x_{|S|}]$ is the optimal solution resulting in maximum total energy savings E_t^{max}, then for any $0 < \rho < 1$ and $x_{max} = (\lceil \frac{1}{\rho \cdot C(G)} \rceil)^2$, there exists $\tilde{x}' = [x'_1 x'_2 ... x'_{|S|}]$ with $\forall i \in 1..|S| : 1 \leq x'_i \leq x_{max}$, for which the total energy savings are greater or equal to $(1-\rho) \cdot E_t^{max}$.*

Proof. First we assume that $x_{i-1} \leq x_{max}$ and $x_{i+1} \leq x_{max}$ and show how we can replace x_i of the solution $\tilde{x}$ by $x'_i \leq x_{max}$:

$$x'_i = \begin{cases} x_i & : \; x_i \leq x_{max} \\ m \cdot n & : \; x_i > x_{max}, x_g \leq m \cdot n \leq x_{max} \\ \max(m, n) & : \; x_i > x_{max}, m \cdot n > x_{max} \\ \lceil \frac{x_g}{m \cdot n} \rceil \cdot m \cdot n & : \; x_i > x_{max}, x_g > m \cdot n \end{cases}$$

where $m = \gcd(x_{i-1}, x_i)$, $n = \gcd(x_i, x_{i+1})$. The value x_g is given by $x_g = \lceil \frac{1}{\rho \cdot C(G)} \rceil$, and $x_{max} = {x_g}^2$.

As before, Type-2 process energy savings are independent of the values of x_i. For Type-1 processes the choice of x_g and the fact that x'_i either is equal to x_i or has a value $\geq x_g$ implies that the energy savings of $\tilde{x}'$ are bounded from below by $(1-\rho)$ multiplied with the energy savings of $\tilde{x}$ (Lemma 2.3).

Therefore, in order to prove the lemma, it is sufficient to show that $lcm(x_{i-1}, x_i) \geq lcm(x_{i-1}\prime, x_i\prime)$ and $lcm(x_i, x_{i+1}) \geq lcm(x_i\prime, x_{i+1}\prime)$.

It is easy to see that it holds for $x_i \leq x_{max}$. For $x_i > x_{max}$ we have:

i. if $m \cdot n > x_{max}$, then $x'_i = \max(m, n)$, and we can assume $x'_i = m$ without loss of

generality. Then,

$$lcm(x_{i-1}, x'_i) = \frac{x_{i-1} \cdot x'_i}{\gcd(x_{i-1}, x'_i)} =$$

$$\frac{x_{i-1} \cdot \not{m}}{\not{m}} = x_{i-1} < \max(x_{i-1}, x_i) \leq lcm(x_{i-1}, x_i)$$

$$lcm(x'_i, x_{i+1}) \leq lcm(x_i, x_{i+1}) \Leftrightarrow$$

$$\frac{m \cdot \not{x}_{i+1}}{\gcd(m, x_{i+1})} \leq \frac{k \cdot m \cdot \not{x}_{i+1}}{\gcd(x_i, x_{i+1})} \Leftrightarrow$$

$$\frac{m}{\gcd(m, l \cdot n)} \leq \frac{k \cdot m}{n} \Leftrightarrow \frac{m \cdot n}{\gcd(m, l \cdot n)} \leq k \cdot m = x_i$$

which is always true because

$$\frac{m \cdot n}{\gcd(m, k \cdot n)} \leq \frac{m \cdot n}{\gcd(m, n)} \leq lcm(m, n) \leq x_i$$

as x_i is a common multiple of both m and n. In this case $m > x_g$ as $m \cdot n > x_{max} = {x_g}^2$ and $m < x_{max}$ as $m = \gcd(x_{i-1}, x_i)$, with $x_{i-1} \leq x_{max}$.

ii. if $x_g \leq m \cdot n \leq x_{max}$, then $x'_i = m \cdot n$ and

$$lcm(x_{i-1}, x'_i) = \frac{x_{i-1} \cdot m \cdot n}{\gcd(x_{i-1}, m \cdot n)} \leq \frac{x_{i-1} \cdot x'_i}{m} \leq \frac{x_{i-1} \cdot x_i}{m} \leq lcm(x_{i-1}, x_i)$$

The same way we can prove $lcm(x_{i+1}, x'_i) < lcm(x_{i+1}, x_i)$

iii. if $m \cdot n < x_g$, then $x'_i = \lceil \frac{x_g}{m \cdot n} \rceil \cdot m \cdot n$

In this case:

$$lcm(x_{i-1}, x'_i) = \frac{x_{i-1} \cdot x'_i}{\gcd(x_{i-1}, x'_i)} \leq \frac{x_{i-1} \cdot x'_i}{m} \leq \frac{x_{i-1} \cdot x_i}{m} \leq lcm(x_{i-1}, x_i)$$

The same way we can prove that $lcm(x'_i, x_{i+1}) \leq lcm(x_i, x_{i+1})$.

The proof so far works for the case of a single value $x_i > x_{max}$ with x_{i-1} and x_{i+1} less or equal to x_{max}. If instead of a single value x_i there is a number of consecutive values $x_{l+1}, ..., x_{r-1}$ that are greater than x_{max} with $x_l, x_r \leq x_{max}$, we follow the procedure above using x_l as x_{i-1} and x_r as x_{i+1}. The value x'_i given by the procedure is the value of all $x_{l+1}, ..., x_{r-1}$. From the above we know that $lcm(x_l, x'_{l+1}) \leq lcm(x_l, x_{l+1})$ and $lcm(x'_{r-1}, x_r) \leq lcm(x_{r-1}, x_r)$. For all other cross edges with $l < k < r$: $lcm(x'_k, x'_{k+1}) = x'_i \leq x_{max} \leq lcm(x_k, x_{k+1})$ since both x_k and x_{k+1} are greater than x_{max}. Values x_l and x_r always exist as $x_0 = x_{|S|+1} = 1$. □

Multirate Graphs. For multirate graphs the buffer sizes depend on the q values. Using a similar approach as for unirate graphs could result in describing x_{max} as a function of q. The q values though can grow exponentially with the input graph [73] and, therefore, a more general method is needed to derive x_{max}.

From Lemma 1 we can derive a bound on the energy savings that can be achieved. Let $E_s(\infty)$ be the sum of the savings for Type-1 processes when x goes to infinity, and $E_s(1)$ when $x = 1$. Also let $E_p(1)$ be the value of the energy penalty when $x = 1$. Let y_i^{max} be the minimum value, for which the energy penalty becomes $E_p^{(i,i+1)}(e, y_i^{max}) \geq E_s(\infty) - E_s(1) + E_p(1)$. We know that increasing y_i to a value greater than y_i^{max} can cause only energy loss, since the savings cannot become greater than $E_s(\infty)$ and $E_p^{(i,i+1)}(e, y_i)$ is increasing with respect to y_i. Therefore, any $y_i > y_i^{max}$ causes an energy penalty that exceeds any energy savings obtained by the Type-1 processes.

Since the energy penalty for all edges is already given (most probably in form of an array of values), binary search can be applied to each of the $(|S| + 1)$ functions $E_p^{(i,i+1)}$ to find y_i^{max}. The binary search procedure can start with a very large value

Y as the maximum value for y that is determined by computational precision limits or area constraints. We know that $y_i = lcm(x_i, x_{i+1}, q_i, q_{i+1})$. Since we also have ${x_{max}}^i \leq lcm({x_{max}}^i, x_{i+1}, q_i, q_{i+1}) = y_i$ and ${x_{max}}^i \leq lcm(x_{i-1}, {x_{max}}^i, q_{i-1}, q_i) = y_{i-1}$, it holds ${x_{max}}^i \leq \min(y_{i-1}, y_i)$. If for each stage i ${x_{max}}^i = \min(y_{i-1}, y_i)$, then $x_{max} = \max_{\forall i}({x_{max}}^i)$ can be chosen as the maximum value for the whole design. Any increase of x above that value for any of the stages causes energy loss compared to the case, in which all x values are 1.

Lemma 2.5. *For the optimal solution $\tilde{x}$ of the multirate problem the following property holds: $\forall i \in 1..|S| \ : \ 1 \leq x_i \leq x_{max}$.*

Proof. Suppose $\tilde{x_{ini}} = [1\ 1\ ...1]$. Suppose that $\tilde{x}$ is the optimal solution and $\exists x_i \in \tilde{x} \ : \ x_i > x_{max}$. Then from the above discussion it holds $E_t(\tilde{x}) < E_t(\tilde{x_{ini}})$, which is a contradiction. □

This method can be applied to a unirate graph as well, and, therefore, we use it in conjunction with the approaches for unirate graphs described above. We use the minimum of the two x_{max} values produced. The running time of the binary search method described above is $O(|S| \cdot \log Y)$.

2.5. Dynamic Programming Solution

In this section we describe a dynamic programming algorithm which can determine the x values for maximum energy savings given a quality metric. The algorithm is needed because the size of the solution space is still large after bounding the x values with x_{max}. Exhaustive search requires $O({x_{max}}^{|S|})$ steps to find the x values for maximum energy savings.

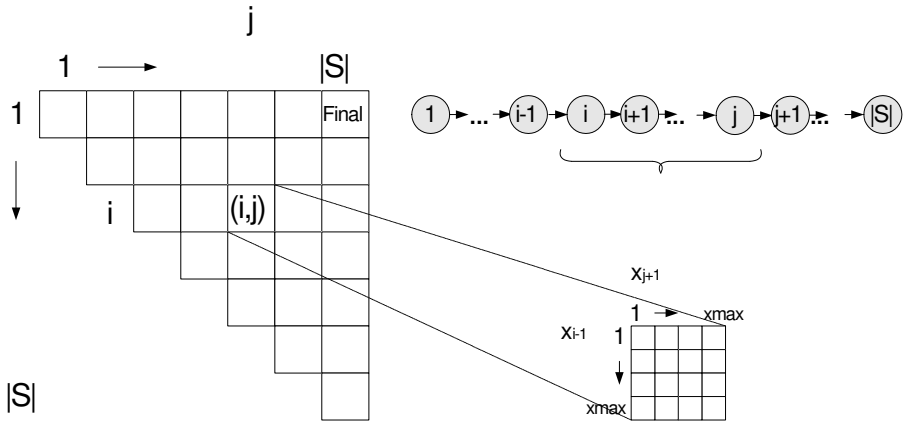


Figure 2.6. Dynamic Programming Algorithm. The solution for subchain (i, j) depends only on values x_{i-1} and x_{j+1}. In a ${x_{max}}^2$ array the best configuration of (i, j) is stored for each value combination of x_{i-1} and x_{j+1}.

In Figure 2.8 the algorithm can be seen. The inputs are the graph which is partitioned in pipeline stages and a quality metric in case the graph is unirate. After initialization, the algorithm determines x_{max} using the procedures described in Section 2.4. The purpose of the rest of the algorithm is to solve independently the problem for each subchain and combine the solutions to find the optimal solution for the chain-structured graph.

The intuition behind the DP solution is that the values x_{i-1} and x_{j+1} are the only external values that can affect the optimal solution for a subchain from stage i to stage j. More specifically, if $i, ..., j$ is a subchain with $1 \leq i < j \leq |S|$, the best configuration for this subchain, i.e. the vector of x values $[x_i, ..., x_j]$ that provides maximum energy savings, depends only on the x values of the stage exactly before the subchain, i.e., x_{i-1}, and the stage after the subchain, i.e., x_{j+1}, (Figure 2.6). Therefore, an ${x_{max}}^2$ matrix can be constructed storing the maximum energy savings that can be obtained for that subchain for each value of the pair (x_{i-1}, x_{j+1}). Such a matrix can gradually be built for all possible subchains of the problem. This array is denoted as $e_s[|S|][|S|][x_{max}][x_{max}]$

in the algorithm of Figure 2.8. As an example, element $e_s[i, j, x_{i-1}, x_{j+1}]$ holds the best configuration for subchain starting at stage i and ending at stage j, when the x value for stage $i-1$ is x_{i-1} and for stage $j+1$ it is x_{j+1}.

After finding x_{max} the algorithm starts by creating the e_s array for subchains of length 0. The entries filled during this phase are the ones on the main diagonal of the simplified array e_s of Figure 2.6. For each $e_s[i][i]$ the ${x_{max}}^2$ matrix is built from the energy savings for stage i and the energy penalty of both cross edges $(i-1, i), (i, i+1)$ for that stage. In the second phase the algorithm fills the entries for subchains with two elements. Finally, in the third phase the energy savings for all remaining subchains are found. The reason for the separate treatment of subchains with two and more than two elements is to make sure that the energy penalty for the same cross edge is not taken twice into account. The maximum energy savings for the whole graph are stored at position $e_s[1][|S|][1][1]$. This entry represents the whole chain with $x_0 = x_{|S|+1} = 1$. As mentioned before, we assume that stages s_0 and $s_{|S|+1}$ are external and we have no control over them. Therefore, their x values remain 1. Array $x_{best}[|S|, |S|, x_{max}, x_{max}]$ stores the decision taken at each step and information necessary to retrieve the optimal solution.

The algorithm searches all possible values from 1 to x_{max} for x, at each subproblem and, therefore, it solves each subproblem optimally. Moreover, since the subproblems are independent, the algorithm finds the solution with the maximum total energy savings for all $1 \leq x_i \leq x_{max}$.

At each step the algorithm computes the energy savings and energy penalty using functions E_s and E_p. The function for the energy savings can be implemented as described in Section 2.4. More specifically, during initialization, i.e. InitEs(G) step, we can find the

Application	**CD-to-DAT (multirate,#stages=3)**			**K-means (unirate, #stages=10)**			**K-means (unirate, #stages=3)**		
Input Rate	50%	25%	12.5%	11.1 %	8.33%	6.67%	33.33 %	25%	12.5%
Alg. Exec. (secs)	2.97	2.97	2.98	144.39	29.26	3.37	5.79	0.7	0.06
x_{max}	71	71	71	169	100	49	100	49	16
Savings Increase	15.17%	5.25%	2.27%	N/A	107.31%	6.71%	900.38%	24.25%	0%

Table 2.2. Experimental Results for several input rates. The input rates are expressed as a percentage of the worst case input rate. The increase in energy savings is "N/A" when the energy savings of power gating with x=1 for all stages are 0.

energy savings for the Type-2 processes, which are independent of x and, therefore, we do not need to recompute them during the iterations of the algorithm. For Type-1 processes of each stage s we can use the following formula to find the energy savings for a specific x

$$E_s(x) \;=\; \sum_{\forall v \in G_s} L_s \cdot \Delta\mathrm{P}(v) \;-\; \frac{x-1}{x} \sum_{\forall v \in G_s} l(G_s) \cdot \Delta\mathrm{P}(v) \;-\; \frac{1}{x} \sum_{\forall v \in G_s} (l(v) \cdot \Delta\mathrm{P}(v) + E_{sm}(v))$$

It is clear from the equation above that all summations can be computed during the initialization step (InitEs). Then E_s can be computed in constant time for each new value of x. It is assumed that the functions E_p are given by the user in the form of an array and, therefore, the energy penalty for a pair of x values can be returned in constant time. Consequently, the algorithm's complexity is $O(|S|^3 \cdot {x_{max}}^3 \;+\; |S| \cdot \log Y)$ and its memory space requirements are $O(|S|^2 \cdot {x_{max}}^2)$.

Theorem 2.1. *The solution found by the dynamic programming algorithm produces total energy savings $E_t^{alg}(x_{max})$, which are at least $(1-\rho) \cdot E_t^{max}$, if the energy penalty at the cross edges $(i, i+1)$ is an increasing function of both x_i, x_{i+1} and x_{max} is given by $x_{max} = \lceil \frac{1}{\rho \cdot C(G)} \rceil$.*

Proof. From Lemma 3, we know that there is at least one solution for which $\forall i, 1 \leq x_i \leq x_{max}$. for which $E_t \geq (1-\rho) \cdot E_t^{max}$. Since the algorithm produces the maximum energy savings for all solutions with $1 \leq x_i \leq x_{max}$, $E_t^{alg} \geq E_t \Rightarrow E_t^{alg} \geq (1-\rho) \cdot E_t^{max}$. □

Theorem 2.2. *The solution found by the dynamic programming algorithm produces total energy savings $E_t^{alg}(x_{max})$, which are at least $(1-\rho) \cdot E_t^{max}$, if the energy penalty at the cross edges $(i, i+1)$ is an increasing function of $lcm(x_i, x_{i+1})$ and x_{max} is given by $x_{max} = {x_g}^2$, $x_g \geq \lceil \frac{1}{\rho \cdot C(G)} \rceil$.*

Proof. Can be proven similarly to Theorem 1 using Lemma 4. □

Theorem 2.3. *The solution found by the dynamic programming algorithm produces total energy savings $E_t^{alg}(x_{max}) = E_t^{max}$, if the energy penalty at the cross edges $(i, i+1)$ is an increasing function of $lcm(x_i, x_{i+1}, q_i, q_{i+1})$ and x_{max} is given by the binary search procedure described above for multirate graphs.*

Proof. Let $\tilde{x}$ be the optimal solution. Then $\forall i$ with $1 \leq i \leq |S|$, it holds that $1 \leq x_i \leq x_{max}$ (Lemma 5). Since the algorithm finds the optimal solution within that space, it finds the optimal solution to the problem. □

2.6. Experimental Results

The algorithm was implemented as a C++ program taking consistent graphs as an input and determining the x values for each pipeline stage. For the experiments we normalized the power of all components using the static power of the 32-bit latch. The static power of 32-bit output multipliers was set to 25 and the 32-bit cla adders 4 times that of the latch. The static power of the decoding logic for the channels was considered

at the same level as the static power of the latches. On the channels the dynamic power increase with x, caused by the extra wiring and control, was considered 50% of the static power increase. The switching mode overhead was considered equal with the energy savings obtained by 10 cycle time slack.

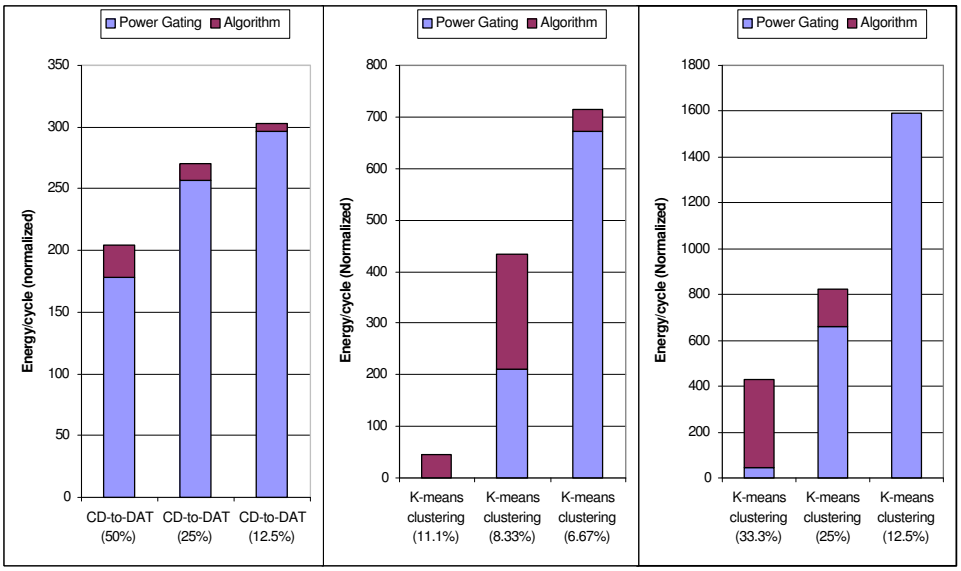


Figure 2.7. Energy savings obtained by our technique and power gating (red and blue) compared to the savings obtained by applying only power gating (blue) for several input rates of 3 applications: CD-to-DAT (left), 10-stage pipeline K-means (middle), 3-stage pipeline K-means (right). The input rates are expressed as a percentage of the worst case input rate below the name of the application.

We applied the algorithm on three pipelined architectures. The first is the CD to DAT sample-rate conversion graph adopted from [68]. Each of the 3 SDF actors of the multirate graph was considered a pipeline stage. The FIR filters were assumed to be 4-tap filters implemented with multipliers. Upsampling, filtering, and downsampling units were considered independent processes forming together one stage. So, in total there were 3 stages in the first level of hierarchy executing in parallel. The second application was the

unirate 10-stage pipelined K-means clustering with euclidean distances adopted from [31]. The third was a 3-stage pipelined architecture for K-means clustering. In the latter case the first two, intermediate four, and last four pipeline stages of the 10-stage pipelined K-means were merged to form a 3-stage pipeline. Figure 2.7 shows the energy savings obtained by our algorithm compared to the energy savings taken by applying power gating only (all x values equal to 1). In all graphs a mode switch occurred for a process only if the energy savings obtained by the switch exceeded the energy overhead E_{sm}.

For each application we tried several input rates. As stated in the introduction we assume that the set of input rates is predefined and changes in the input rates happen with a low frequency (e.g. user-controlled). For higher input rates the idle time in each complete cycle is shorter. Therefore, the energy savings obtained by power gating (all x values equal to 1) are low or zero. For these cases applying the proposed technique has a significant impact as seen from the last row of Table 2.2. As the input rate is reduced, mode transitions occur less often. The energy consumption because of the mode switch overhead becomes less significant and, consequently, the additional savings obtained by the proposed technique decrease.

For the 3-stage pipelined K-means gains are produced in higher input rates than for the 10-stage pipelined architecture. The reasons for this are that more Type-1 processes are sharing the penalty paid on the cross edges of one stage, and that the slack for each process is increased because the latency of each stage ($l(G)$) is longer. In Table 2.2 the results are shown. Finally, in Table 2.3 the effect of the ρ value on running-time can be seen. In this case the energy savings are the same for different ρ values. However, a higher

Application Input Rate	**10-stage K-means** 6.67%		**3-stage K-means** 25%	
ρ	0.90	0.95	0.90	0.95
Alg. Exec. Time(sec)	3.37	351	0.7	44.1
x_{max}	49	225	49	196
Increase in En. Savings	6.71%	6.71%	24.25%	24.25%

Table 2.3. Effect of the ρ value on running-time.

ρ value offers a guarantee for the proximity to the optimal solution, whereas a lower ρ value results in a shorter running-time.

2.7. Summary

In this chapter we presented an approach to reduce energy consumption using power gating. An analysis framework was presented and a theoretical bound on the number of consecutive iterations was derived for chain-structured pipelines. An algorithm was developed that can give an optimal solution for the total energy savings. In the next chapter we show how we can improve the performance of an application described as an SDF graph.

Algorithm DP-for x values
Input: A chain structured SDF graph $G = (S, E)$ representing the pipeline stages, a quality metric ρ which will be used if G is unirate, and functions $E_p^{(i,i+1)}(x_i, x_{i+1})$ returning the energy overhead for cross edges between stages i and i+1.
Output: Two arrays $x_{best}[|S|, |S|, x_{max}, x_{max}]$ and $e_s[|S|, |S|, x_{max}, x_{max}]$ from which the optimal solution can be extracted.

```
InitEs(G);
x_max ← DetermineXmax(G,ρ);
for i ← 1 to |S| do // main diagonal d = 0
  for x_{i-1} ← 1 to x_max do
    for x_{i+1} ← 1 to x_max do
      for x_i ← 1 to x_max do
        e_s^new ← E_s^i(x_i) − E_p^{(i-1,i)}(x_{i-1}, x_i) − E_p^{(i,i+1)}(x_i, x_{i+1})
        if (e_s[i, i, x_{i-1}, x_{i+1}] < e_s^new) then
          e_s[i, i, x_{i-1}, x_{i+1}] ← e_s^new ; x_best[i, i, x_{i-1}, x_{i+1}] ← x_i
for i ← 1 to |S| − 1 do // init step for d = 1
  for x_{i-1} ← 1 to x_max do
    for x_{i+2} ← 1 to x_max do
      for x_node ← 1 to x_max do
        // x_node represents x_i and x_{i+1} in this loop
        e_s^new1 ← e_s[i, i, x_{i-1}, x_node] + E_s^{i+1}(x_node) − E_p^{(i+1,i+2)}(x_node, x_{i+2})
        e_s^new2 ← e_s[i + 1, i + 1, x_node, x_{i+2}] + E_s^i(x_node) − E_p^{(i-1,i)}(x_{i-1}, x_node)
        e_s^new ← max(e_s^new1, e_s^new2)
        node ← (e_s^new1 > e_s^new2)
        if (e_s[i, i + 1, x_{i-1}, x_{i+2}] < e_s^new) then
          e_s[i, i + 1, x_{i-1}, x_{i+2}] ← e_s^new ; x_best[i, i + 1, x_{i-1}, x_{i+2}] ← (node, x_node)
for d ← 2 to |S| − 1 do //diagonal count
  for i ← 1 to |S| − d do
    j ← i + d
    for k ← 1 to j − i − 1 do
      for x_{i-1} ← 1 to x_max do
        for x_{j+1} ← 1 to x_max do
          for x_{i+k} ← 1 to x_max do
            e_s^new ← e_s[i, i + k − 1, x_{i-1}, x_{i+k}] + E_s^{i+k}(x_{i+k}) + e_s[i + k + 1, j, x_{i+k}, x_{j+1}]
            if (e_s[i, j, x_{i-1}, x_{j+1}] < e_s^new) then
              e_s[i, j, x_{i-1}, x_{j+1}] ← e_s^new ; x_best[i, j, x_{i-1}, x_{j+1}] ← (i + k, x_{i+k})
Return x_best, e_s;
```

Figure 2.8. Pseudocode describing the dynamic programming algorithm.

CHAPTER 3

Performance Optimization of Synchronous Data Flow Graphs

3.1. Introduction

In Chapter 2 we introduced Synchronous Dataflow (SDF) Graphs and described an approach to reduce energy consumption of system-level pipelines described as chain-structured SDF Graphs. In this chapter we show a retiming algorithm to optimize the performance of any application that can be described as a general SDF Graph.

SDF Graphs are considered a useful way to model DSP applications [55]. This is because in most cases the portions of DSP applications, where most of the execution-time is spent, can be described by processes or actors with constant rates of data consumption and production. Moreover, efficient memory and execution-time minimization algorithms have been developed for SDF graphs [12, 35].

Retiming has been used widely in the past to optimize the cycle time or resources of gate-level graph representations [57, 60, 88]. A lot of work has also been done on extending retiming on SDF graphs [70, 91]. More specifically, retiming has been proposed to facilitate vectorization [90] and to minimize the cycle length of these graphs [70].

Govindarajan and Gao have proposed an algorithm to determine a non-blocking schedule for an SDF graph for maximum throughput [35]. However, there are design cases in which a non-blocking schedule is not feasible. This happens when a part of the application's behavior is determined dynamically at run-time, or when some of the application's

tasks share resources with higher-priority tasks. These tasks are normally executed on a programmable processor, while the computationally expensive part of the application is run on dedicated resources, has predictable execution time, and is conveniently modeled as an SDF. In case there are data dependencies between the SDF actors and the tasks executed on the programmable processors, a non-blocking schedule may not be feasible. Then a blocking schedule for the SDF is necessary and the blocking schedule with the minimum cycle length is equivalent to minimum latency of the static part of the application (Figures 3.1,3.2).

O'Neil and Sha proposed an algorithm to reduce the clock cycle of a graph below a threshold using retiming [70]. We propose an optimal algorithm for retiming SDF graphs. The purpose is to minimize the length of the complete cycle of a SDF graph. Two versions of the algorithm are shown. Both produce better results than any existing algorithm. Moreover, the second one is orders of magnitude faster than O'Neil's algorithm.

In Sections 3.2 and 3.3 we present the basic properties of SDF graphs. An optimal algorithm for minimizing the period of a blocking schedule for an SDF is described in Section 3.3. Then in Section 3.4 the first version of the retiming algorithm is presented and in Section 3.5 its correctness is proven. An improved version of this algorithm is described in Section 3.6. Finally in Sections 3.8 and 3.9 the experimental results and conclusions are presented.

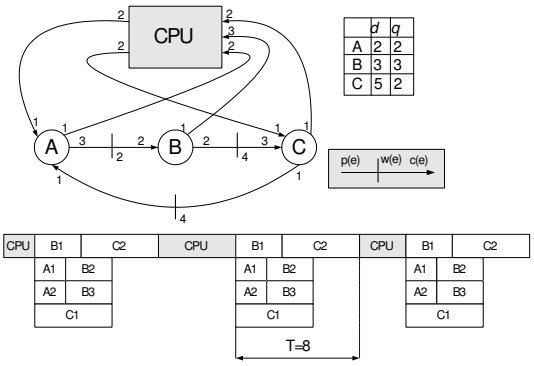


Figure 3.1. Initial SDF schedule.

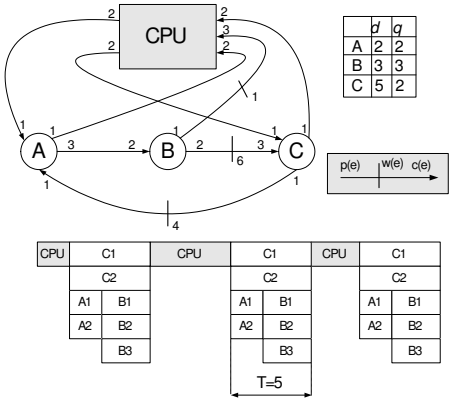


Figure 3.2. Improved SDF schedule.

3.2. Synchronous Data Flow Graphs

A short description of Synchronous Data Flow graphs is given in Section 2.2. Here we summarize their basic properties that we are going to use in this chapter. For a more detailed description of the SDF properties the user should refer to the literature [55].

An SDF graph is a directed graph $G = (V, E, d, p, c, w)$, in which $d : V \rightarrow \mathbb{R}^+$ is a function giving the execution delay of a node, and $p, c, w : E \rightarrow \mathbb{N}$ are functions which give the production rate, consumption rate, and initial number of tokens of each edge. Table 3.1 lists all these symbols with their definitions.

In this work only live and consistent SDFs are considered, which can execute without deadlock and with finite memory for an infinite number of times. A necessary condition for a graph to be consistent is

$$P0 \triangleq (\forall (u, v) \in E : q_u \cdot p\,(u, v) = q_v \cdot c\,(u, v))$$

We assume that computing resource constraints for a specific actor are captured by loops, i.e. edges with the same node as head and tail. The number of delays of each loop determines the number of actor executions that can occur concurrently. The production and consumption rate of the loop are set to 1.

If an actor carries state there is a loop on the node representing the actor with $p = c = 1$. The number of delays on the node denote the distance of the dependency in terms of number of executions. As an example for an FIR filter the number of delays is 1, since each instance execution depends on the previous one. Instances of the same SDF node can execute concurrently as long as they do not violate self-dependencies and other constraints imposed by the structure of the graph. The ordering of the produced output tokens and consumed input tokens is taken care by the control mechanism of the edge.

The period for a gate-level graph [57] is defined by the longest path in the graph. In that time all nodes must be executed exactly once. In a consistent SDF graph different nodes can have different average invocation rates. The solution with the minimum positive integers to the state equations gives the number of times each node needs to be executed in a system period or complete cycle of the graph. In a blocking schedule complete cycles of the graph cannot be overlapped. Therefore, the length of the complete cycle can be considered the period of the graph. We consider only blocking schedules for SDF graphs.

3.3. Retiming Properties for SDF Graphs

3.3.1. Node r Values

In gate-level retiming [57] the $r(v)$ value of a node v denoted the number of registers moved from each output edge to the input edges of v.

Retiming in SDF is applied on instance executions of a node v. Each instance execution consumes $c(u,v)$ tokens from each incoming edge (u,v) and produces $p(v,z)$ tokens to each outgoing edge (v,z). Increasing $r(v)$ by one is equivalent to "canceling" the execution of one instance of v. Therefore, the outgoing edges have their weights decreased by $p(v,z)$ and the incoming edges have their weights increased by $c(u,v)$. For $(u,v) \in E$ the number of delays $w_r(u,v)$ after each retiming step is given by

$$w_r(u,v) = w(u,v) + r(v) \cdot c(u,v) - r(u) \cdot p(u,v) \tag{3.1}$$

Since for any valid retiming the final number of delays on each edge must be non-negative,

$$P1 \triangleq (\forall (u,v) \in E : w_r(u,v) \geq 0)$$

must hold for any valid retiming.

It can be easily proven that any retiming solution with integer values satisfying the above properties defines a new graph which belongs to the reachable space of the initial graph [91].

3.3.2. Computing the Max-Length Path

The longest path computation in previous works was done either on the EHG (Equivalent Homogeneous Data-Flow Graph) [70] or the precedence graph [35]. We show a way to compute the longest path by using the original SDF graph.

If the repetitions vector of a graph is $\mathbf{q} = [q_1,\ q_2,\ \ldots,\ q_{|V|}]$, then each system iteration (or complete cycle of the graph) will include q_v executions of SDF node v. We call these q_v instance executions of v.

We know that since the edges implement FIFO channels, there exists an implicit partial order for the executions of the instances of v. For each node v with $k \in N$ and $1 < k \leq q_v$:

$$t(v, k-1) \leq t(v, k) \tag{3.2}$$

where $t(v, k)$ is the arrival time at the inputs of the instance k of node v. In order to find the maximum longest path of one complete cycle of the graph it is enough to find the $\max_{\forall v \in V}(t(v, q_v) + d(v))$.

A recursion equation we can use for this purpose is

$$t\left(v, k\right) = \max_{\forall (u,v) \in E} \left(t\left(u, l\right) + d\left(u\right)\right) \tag{3.3}$$

where the l instance of node u is given by

$$l = \lceil \frac{k \cdot c(u, v) - w_r(u, v)}{p(u, v)} \rceil \tag{3.4}$$

The above equations define an ASAP scheduling. Instance k of node v is executed immediately after all the necessary tokens are present in the input FIFO channels. The instances, on which k depends on, are found for each edge incoming to v by (3.4). For the kth instance to be executed $k \cdot c(u, v)$ tokens must have been available on each channel $(u, v) \in E$. The lth instance of u node is the first instance that guarantees that the $w_r(u, v)$ already present tokens together with the $l \cdot p(u, v)$ produced in the current complete cycle reach this number.

In (3.4), l can be less than or equal to zero. This means that the kth instance of v node depends on the $q_u + l$ instance of the previous complete cycle. We require

$$\forall u \in V, \forall l \in \mathbb{Z} \ : \ (l \leq 0 \Rightarrow t(u,l) + d(u) = 0)$$

This property makes the scheduling blocking. As instance k cannot start execution before time 0, when the current complete cycle begins, this property prevents complete cycles from overlapping.

We can make (3.3) weaker by replacing equality. Then the following property needs to hold

$$P2 \triangleq \left(\begin{array}{l} \forall v \in V, \forall (u,v) \in E, \forall k \in \mathbb{Z} : \\ \qquad \text{Let } l \triangleq \lceil \frac{k \cdot c(u,v) - w_r(u,v)}{p(u,v)} \rceil \text{ in} \\ \qquad\qquad (1 \leq k \leq q_v) \Rightarrow (\, t(v,k) \geq t(u,l) + d(u) \,) \end{array} \right)$$

The blocking schedule property is equivalent to

$$P3 \triangleq \big(\forall v \in V, \forall k \in \mathbb{Z} \ : \ (k < 1) \ \Rightarrow \ (\, t(v,k) = -d(v) \,) \big)$$

$P2$ and $P3$ are more general and hold for any valid blocking scheduling instead of an ASAP blocking scheduling of the instance nodes.

3.3.3. Optimal Period

For the period T of a blocking schedule of an SDF graph, it must hold

$$\forall v \in V, \forall k \in \mathbb{Z} \ : \ (1 \leq k \leq q_v) \Rightarrow t(v,k) + d(v) \leq T$$

Because of (3.2) the following property is necessary and sufficient

$$P4 \triangleq (\forall v : t\,(v, q_v) + d\,(v) \leq T)$$

for T to be the period of the schedule. The period T can be considered as a function of the retiming vector $\mathbf{r} = [r_1,\ r_2,\ \ldots,\ r_{|V|}]$, which ranges over $\mathbb{R}^+$. For the optimal period $T(\mathbf{r})$ of a blocking schedule the following property holds

$$P5 \triangleq (\forall \mathbf{r}' : T(\mathbf{r}) \leq T(\mathbf{r}'))$$

3.3.4. Problem Formulation

Given a consistent SDF graph $G = (V, E, d, p, c, w)$ find a retiming $\mathbf{r}$ and minimum complete cycle length $T(\mathbf{r})$ that satisfy properties P1-P5.

3.4. Retiming Algorithm

In this section the retiming algorithm is described. The algorithm can be seen in Figures 3.3,3.4, and 3.5.

The algorithm uses procedures *get_t* and *init_t* to find the arrival times. From $P4$ we note that only the last (qth) instance of each node is important to find the period of the complete cycle. Therefore, it is sufficient to compute $\forall v \in V, t(v, q_v)$ and the arrival times of their dependencies. Procedure *get_t* achieves that by recursively calling itself on the dependencies of an instance node. Therefore, the procedure avoids computing the arrival nodes of instance nodes that cannot change the arrival time of (v, q_v) for any $v \in V$.

Symbol	Definition
q_v	number of executions (instances) of node v in one complete cycle
$d(v)$	execution time for each instance of node v
$p(u,v)$	number of tokens produced on edge (u,v) as a result of an execution of node u
$c(u,v)$	number of tokens of edge (u,v) consumed as a result of an execution of node v
$w(u,v)$	number of initial tokens (delays) on edge (u,v) in the input graph
$r(v)$	retiming value for node v
$\mathbf{r}$	vector $1 \times \|V\|$ containing the retiming values for all nodes of the graph
$w_r(u,v)$	number of delays on edge (u,v) after $\mathbf{r}$ has been applied to the graph
$t(v,k)$	arrival time for the instance k of node v, the time when the tokens for the kth instance are available $\forall(u,v) \in E$
T	latency of a complete cycle of the SDF graph, equals the period of a blocking schedule
$\mathbb{N}$	the set of natural numbers including 0
$\mathbb{Z}$	the set of integers
$\mathbb{R}^+$	the set of positive real numbers, which are greater than 0

Table 3.1. Definition of the symbols used in this chapter. Some of these symbols have already been defined in Chapter 2, but we repeat them here for the reader's convenience.

The procedure avoids recomputation of the arrival times of the same instance nodes by maintaining an array $t[|V|, q_v]$. This array holds the arrival times of the nodes already computed. Initially, the entries of this array are set to $-\infty$ (could be any illegal number) by procedure *init_t*. Any computed arrival time is stored in the array. Arrival times are only computed if the value in the array is $-\infty$, or else the already computed value is returned.

Property $P3$ is preserved by the first two lines of procedure *get_t*.

By implementing *get_t* as a memory function working directly on the SDF, the expensive construction of an EHG or a precedence graph is avoided.

Restricting the computation of the arrival times to only those instances that can affect the period has an effect on the properties discussed above. More specifically, it is equivalent to relaxing P2 to be valid only for the the qth instance of each nodes and its dependencies. It is easy to show though that for any result using these arrival times we can obtain arrival times for all node instances that validate P2 using an ASAP algorithm. For efficiency reasons, however, the algorithm does not compute the arrival times for all nodes in each iteration. Predicate P2 can be replaced by a weaker predicate $P2'$. $P2'$ is true, whenever for the arrival times obtained there exists an algorithm S to compute the rest of the node instance arrival times, such that P2 can be validated

$$P2' \triangleq (\exists S : P2)$$

With the arrival times obtained by get_t, predicate $P2'$ is true.

The algorithm in Figure 3.5 starts by initializing the memory function elements for all arrival times to $-\infty$. Then it sets $\forall v, r(v) = 0$ and computes the arrival times for all (v, q_v). After finding the $maxt = \max(t(v, q_v) + d(v))$, it sets $T_{step} = maxt$ and enters the while loop. In each iteration of the while loop, $r(v_n)$ is increased by 1, where v_n is the node for which $maxt = t(v_n, q_{v_n}) + d(v_n)$ in the previous iteration. If $maxt < T_{step}$, then T_{step} becomes equal to $maxt$ and the algorithm tries to find another $\mathbf{r}$ with $T(\mathbf{r}) < T_{step}$.

Each time an r-value changes the algorithm recomputes the arrival times using the memory function. This way after each r change the algorithm keeps predicates $P2'$ and $P3$ invariant. $P4$ is always satisfied by $maxt$ and the current iteration's $\mathbf{r}$. Therefore, it is satisfied by $(\mathbf{r^o}, T_{step})$ when the algorithm exits.

In order, to understand the reason $P1$ is kept invariant as well, we have to refer to (3.4). An edge (v, z) can have $w'_r(v, z) < 0$ if before the change of $r(v)$ to $r(v) + 1$, there were $w_r(v, z) < p(v, z)$ tokens. But in that case

$$\begin{aligned} l_v &= \lceil \frac{q_z \cdot c(v,z) - w_r(v,z)}{p(v,z)} \rceil \stackrel{P0}{=} \lceil \frac{q_v \cdot p(v,z) - w_r(v,z)}{p(v,z)} \rceil \\ &\geq \lceil \frac{q_v \cdot p(v,z)}{p(v,z)} \rceil = q_v \stackrel{l_v \leq q_v}{\Rightarrow} \\ l_v &= q_v \end{aligned}$$

But that means that (z, q_z) instance can only start after (v, q_v) has completed execution and, therefore,

$$t(z, q_z) + d(z) > t(z, q_z) \geq t(v, q_v) + d(v) = maxt \tag{3.5}$$

which is a contradiction. $P1$ is also an invariant of the algorithm. Only property $P5$ may not be true after initialization and becomes true upon termination of the while loop algorithm, as proven in the next section.

```
proc init_t(v,k)
   for each v ∈ V
      for each k ← 1 to q_v
         t[v,k] ← −∞;
      endfor;
   endfor;
```

Figure 3.3. Procedure for initializing the arrival times.

3.5. Algorithm Correctness

In this section the correctness of the algorithm is proven. Our analysis is restricted to strongly connected graphs. In Section 3.7 it is shown how to extend the approach

```
proc get_t(v,k,r)
    if (k < 1) then
       return −d(v);
    fi;
    if (t[v, k] ≠ −∞) then
       return t[v, k];
    fi;
    maxt ← −1;
    for each (u, v) ∈ E
       l ← ⌈(k·c(u,v) − w_r(u,v)) / p(u,v)⌉;
       t1 ←get_t(u,l)+d(u);
       if (maxt < t1) then
          maxt ← t1;
       fi;
    endfor;
    t[v, k] ← maxt;
    return t[v, k];
```

Figure 3.4. Procedure for getting the arrival time of a node.

to graphs with input/output channels, sources and sinks. Properties of the solution of the problem are proved in Section 3.5.1. Based on these properties we prove the two termination conditions and the correctness of the algorithm (Sections 3.5.2, 3.5.3).

3.5.1. Analysis

In this section we analyze the properties of strongly connected SDF graphs. The main results of this section are Theorem 3.1 and Lemma 3.5. Theorem 3.1 provides a justification on the algorithm's choice for the node to be retimed and is used in the proof of the optimality of the algorithm. Lemma 3.5 describes the nature of the possible solutions. The definitions of a dependency walk and a critical dependency walk provide the necessary concepts for understanding the proofs. Finally, Lemmas 3.1–3.4 are used in the proof of Theorem 3.1.

```
Algorithm SDF Retiming
Input: An SDF graph G = (V, E, d, p, c, w).
Output: A pair (r, T_min) which represents an
      optimal retiming r satisfying minimum
      complete cycle execution time T_min.
   maxt ← 0;
   init_t();
   for each v in V do
      r(v) ← 0; t(v, q_v) ← get_t(v, q_v);
      if t(v, q_v) + d(v) > maxt then
         maxt ← t(v, q_v) + d(v); v_n ← v;
      fi;
   endfor;
   T_step ← maxt;
   while((∃v : r(v) < q_v) ∧ (∄v : r(v) > 2 · q_v · |V|)) do
      r(v_n) ← r(v_n) + 1;
      init_t();
      for each v in V do
         t(v, q_v) ← get_t(v, q_v);
         if t(v, q_v) + d(v) > maxt then
            maxt ← t(v, q_v) + d(v); v_n ← v;
         fi;
      endfor;
      if maxt < T_step then
         r^o ← r;
         T_step ← maxt;
      fi;
   endwhile;
   Return (r^o, T_step);
```

Figure 3.5. Pseudocode describing the first version of the retiming algorithm.

By using the ordered pair (v, l) we denote a node v and the instance number l, for which $1 \leq l \leq q_v$.

Lemma 3.1. *Let (u, v) be an edge in E, $1 \leq l_v \leq q_v$, and*

$$l_u \triangleq \lceil \frac{l_v \cdot c(u,v) - w_r(u,v)}{p(u,v)} \rceil$$

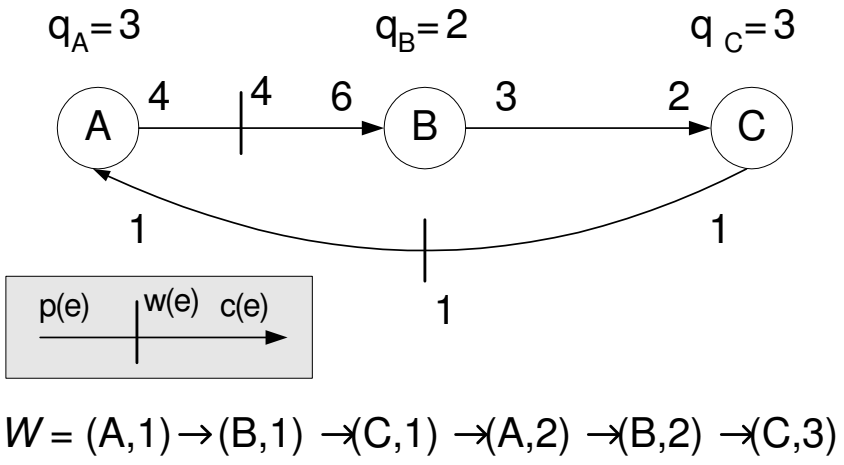


Figure 3.6. An example of a dependence walk.

There is a dependency relation between node instances (u, l_u) *and* (v, l_v)*, if and only if* $1 \leq l_u \leq q_u$.

Proof. If there is a dependency relation, then execution of (v, l_v) can only start in the current complete cycle after the execution of (u, l_u) has completed. Assume $l_u > q_u$, then node instance (u, l_u) belongs to a future complete cycle and (v, l_v) must wait for (u, l_u) execution. This is a contradiction as we assume that the scheduling is blocking and complete cycles do not overlap. Therefore, $l_u \leq q_u$ holds. Now assume $l_u < 1$, then (u, l_u) belongs to the previous complete cycle. When the current complete cycle starts, execution of (u, l_u) has already been completed. Therefore, no dependency exists in the current complete cycle between (u, l_u) and (v, l_v), which is a contradiction. Consequently, $1 \leq l_u \leq q_u$ holds.

Assume that $1 \leq l_u \leq q_u$. Then by Equation 3.4 node instance (v, l_v) has to wait for the completion of (u, l_u) for the necessary tokens to be created in the current complete cycle. Therefore, there is dependency relation between (u, l_u) and (v, l_v). □

Definition 3.1. *A dependence walk*

$$\mathcal{W} = (v_0, l_0) \rightarrow (v_1, l_1) \rightarrow ... \rightarrow (v_n, l_n)$$

is a walk in the SDF graph G in which the execution of (v_i, l_i) *can only start after the execution of* (v_{i-1}, l_{i-1}) *has been completed for all* $i, 0 \leq i < n$.

That means that there is a dependence relation between every two consecutive node instances in the dependence walk.

For each (v_i, l_i) it holds that $t(v_i, l_i) \geq t(v_{i-1}, l_{i-1}) + d(v_{i-1})$. Also note that in $\mathcal{W}$ there can be multiple appearances of the same SDF node with a different label each time (Figure 3.6). That means that there could be (v_i, l_i) and (v_j, l_j) with $v_i = v_j$ and $l_i \neq l_j$. Moreover, in $\mathcal{W}$ an SDF edge may be used multiple times to define a dependency. From now the term walk denotes a dependence walk in the SDF graph.

Definition 3.2. *A critical walk is a walk for which*

$$\forall i : (1 \leq i \leq n) \Rightarrow (t(v_i, l_i) = t(v_{i-1}, l_{i-1}) + d(v_{i-1}))$$

and $t(v_0, l_0) = 0$.

For a critical walk the first node starts exactly at time 0, which is the beginning of the complete cycle. All other nodes start exactly at the time their predecessor in the walk has completed execution.

Lemma 3.2. *Suppose* $\mathcal{W} = (v_0, l_0) \rightarrow ... \rightarrow (v_n, l_n)$ *is a critical walk and* $t(v_n, q_n) + d(v_n) = T(\mathbf{r})$ *for a retiming vector* $\mathbf{r}$*, then for any retiming vector* $\mathbf{r}'$ *for which a dependence walk* $\mathcal{W}' = (v_0, l'_0) \rightarrow ... \rightarrow (v_n, l'_n)$ *exists,* $T(\mathbf{r}') \geq T(\mathbf{r})$ *holds.*

Proof. Since $\mathcal{W}'$ is still a valid walk in the graph, each edge in it specifies a dependency. Therefore, the following property holds

$$\forall i : (\, 0 \leq i < n \,) \Rightarrow (\, t'(v_{i+1}, l_{i+1}) \geq t'(v_i, l_i) + d(v_i) \,)$$

For the first node instance of the walk $t'(v_0, l'_0) \geq 0 = t(v_0, l_0)$. If for k, $t'(v_k, l_k) \geq t(v_k, l_k)$ holds, then for $k + 1$

$$t'(v_{k+1}, l_{k+1}) \geq t'(v_k, l_k) + d(v_k) \geq t(v_k, l_k) + d(v_k) = t(v_{k+1}, l_{k+1})$$

So, by induction $t'(v_n, l_n) \geq t(v_n, l_n)$, which implies $T(\mathbf{r}') \geq T(\mathbf{r})$. □

Lemma 3.3. *Suppose* $\mathcal{W} = (v_0, l_0) \rightarrow ... \rightarrow (v_n, l_n)$ *is a dependence walk. Let* u *be any node, such that* $\forall i \in 1..n : v_i \neq u$*, i.e. none of the node instances in* $\mathcal{W}$ *is a node instance with* u*. Then by increasing the* r *value of* u*,* $\mathcal{W}$ *remains a dependency walk in the graph.*

Proof. If $r(u)$ changes value, the number of weights only on edges, which are incoming or outgoing to u, will change. Since for all consecutive node instances (v_i, l_i) and (v_{i+1}, l_{i+1}) in $\mathcal{W}$, $u \neq v_i$ and $u \neq v_{i+1}$, none of the edges of $\mathcal{W}$ has their $w_r(v_i, v_{i+1})$ modified. Therefore, from Lemma 3.1, the dependence relations between the node instances in $\mathcal{W} = (v_0, l_0) \rightarrow ... \rightarrow (v_n, l_n)$ are preserved. Consequently, $\mathcal{W}$ is still a dependence walk in the graph. □

The following lemma is about the case of increasing the r value of any node of the walk, except the last node. More specifically, it states that if we increase the r value of any node u, such that there exists node instance (u, l_u) in $\mathcal{W}$, but $u \neq v_n$, a walk with the same number of elements and the same nodes will exist in the graph.

Lemma 3.4. *Suppose $\mathcal{W} = (v_0, l_0) \rightarrow ... \rightarrow (v_n, l_n)$ is a dependency walk. Let u be any node in V such that there exists an node instance (u, l_u) in $\mathcal{W}$, but $u \neq v_n$. If the r-value of node u is increased by Δr_u, another dependency walk*

$$\mathcal{W}' = (v_0, l'_0) \rightarrow ... \rightarrow (v_n, l'_n)$$

exists in the graph with

$$l'_i = \begin{cases} l_i & : \quad v_i \neq u \\ l_i + \Delta r_u & : \quad v_i = u \end{cases} \tag{3.6}$$

Proof. We are going to start from node instance $(v_n, l'_n) = (v_n, l_n)$ and walk backwards by induction to prove that $\mathcal{W}'$ exists. For v_n the above relation holds, since $v_n \neq u$ and $l'_n = l_n$ by the way instance node (v_n, l'_n) was chosen.

Suppose that it holds for node instance (v_i, l'_i) with $i \in 1..n$, then we show that it holds for node instance (v_{i-1}, l'_{i-1}), for which v_{i-1} is the node before v_i in $\mathcal{W}$ and l'_{i-1} is instance of node v_{i-1}, on which (v_i, l'_i) depends, as given by (3.4). In the proof we use $(i-1, i)$ to denote the edge (v_{i-1}, v_i).

For v_{i-1} and v_i there are 4 cases:

Case 1: $v_{i-1} \neq u \ \wedge \ v_i \neq u$

In this case $l'_i = l_i$ by the induction assumption and $w_r(i-1, i) = w'_r(i-1, i)$, since $r'(v_{i-1}) = r(v_{i-1})$ and $r'(v_i) = r(v_i)$. Because of (3.4), that implies $l'_{i-1} = l_{i-1}$.

Case 2: $v_{i-1} = u \ \wedge \ v_i \neq u$

In this case $l'_i = l_i$ by the induction assumption and $w'_r(i{-}1, i) = w_r(i{-}1, i){-}\Delta r_u{\cdot}p(i{-}1, i)$. Then

$$\begin{aligned} l'_{i-1} &= \lceil \frac{l'_i \cdot c(i-1,i) - w'_r(i-1,i)}{p(i-1,i)} \rceil \\ &= \lceil \frac{l_i \cdot c(i-1,i) - w_r(i-1,i) + \Delta r_u \cdot p(i-1,i)}{p(i-1,i)} \rceil \\ &= \lceil \frac{l_i \cdot c(i-1,i) - w_r(i-1,i)}{p(i-1,i)} \rceil + \Delta r_u \\ &= l_{i-1} + \Delta r_u \end{aligned}$$

Case 3: $v_{i-1} \neq u \ \wedge \ v_i = u$

In this case $l'_i = l_i + \Delta r_u$ by the induction assumption and $w'_r(i-1, i) = w_r(i-1, i) + \Delta r_u \cdot c(i-1, i)$. Then

$$\begin{aligned} l'_{i-1} &= \lceil \frac{l'_i \cdot c(i-1,i) - w'_r(i-1,i)}{p(i-1,i)} \rceil \\ &= \lceil \frac{(l_i + \Delta r_u) \cdot c(i-1,i) - w_r(i-1,i) - \Delta r_u \cdot c(i-1,i)}{p(i-1,i)} \rceil \\ &= \lceil \frac{l_i \cdot c(i-1,i) - w_r(i-1,i) + \Delta r_u \cdot c(i-1,i) - \Delta r_u \cdot c(i-1,i)}{p(i-1,i)} \rceil \\ &= l_{i-1} \end{aligned}$$

Case 4: $v_{i-1} = u \ \wedge \ v_i = u$

In this case $l'_i = l_i + \Delta r_u$ by the induction assumption. The edge in the SDF graph that models this dependency is a loop (u, u). Since the production rate of such an edge is equal

to the consumption rate, it holds

$$\begin{aligned} w'_r(i-1,i) &= w_r(i-1,i) - p(i-1,i)\cdot\Delta r_u + c(i-1,i)\cdot\Delta r_u \\ &= w_r(i-1,i) \end{aligned}$$

Then

$$\begin{aligned} l'_{i-1} &= \lceil \frac{l'_i \cdot c(i-1,i) - w'_r(i-1,i)}{p(i-1,i)} \rceil \\ &= \lceil \frac{(l_i + \Delta r_u)\cdot c(i-1,i) - w_r(i-1,i)}{p(i-1,i)} \rceil \end{aligned}$$

But because $p(i-1,i) = c(i-1,i)$, it holds that

$$\begin{aligned} l'_{i-1} &= \lceil \frac{l_i \cdot c(i-1,i) - w_r(i-1,i) + \Delta r_u \cdot p(i-1,i)}{p(i-1,i)} \rceil \\ &= l_{i-1} + \Delta r_u \end{aligned}$$

As we note in all cases, the l'_{i-1} as given by (3.4) satisfies (3.6). For $0 \leq i \leq n$, $l'_{i-1} \geq l_{i-1} \geq 1$ holds. Since the increase of the $r(u)$ value should not violate $P1$, $(\forall(u,v) \in E : w'_r(u,v) \geq 0) \Rightarrow (\forall(u,v) \in E \ : \ (u \neq v) \ \Rightarrow \ (w_r(u,v) \geq p(u,v)\cdot\Delta r_u))$. Because of (3.4), this implies

$$\begin{aligned} & \forall i \in 0..(n-1) \ : \\ & \quad \big(\ (\ (v_i = u) \wedge (v_{i+1} \neq u) \ ) \ \Rightarrow \ (\ l_i \leq q_u - \Delta r_u \ ) \ \big) \\ \overset{l'_i = l_i + \Delta r_u}{\Leftrightarrow} \quad & \forall i \in 0..(n-1) \ : \\ & \quad (\ (\ (v_i = u) \wedge (v_{i+1} \neq u) \ ) \ \Rightarrow \ (\ l'_i \leq q_u \ ) \ ) \end{aligned}$$

For the case $v_i = u$ and $v_{i+1} = u$, since it holds that $v_n \neq u$, there exists node v_j such that for all $k \in j..n$, $v_k \neq u$. Suppose $v_{j-1} = u$, then $l'_{j-1} \leq q_u$. That implies that for all i, with $i < j$ and $v_i = u$, $l_i < q_u$ holds. Consequently, for each $v_i = u$, $1 \leq l'_i \leq q_u$, and, therefore, $\mathcal{W}'$ exists. □

Theorem 3.1. *Suppose for a retiming* $\mathbf{r}$ *that* $t(v_n, q_n) + d(v_n) = T(\mathbf{r})$. *If* $\exists\ \mathbf{r}'$ *such that* $T(\mathbf{r}') < T(\mathbf{r})$ *and* $\forall v, r'(v) \geq r(v)$, *then* $r'(v_n) > r(v_n)$.

Proof. Since $t(v_n, q_n) + d(v_n) = T(\mathbf{r})$, there exists a critical walk $\mathcal{W} = (v_0, l_0) \rightarrow ... \rightarrow (v_n, l_n)$ with $t(v_0, l_0) = 0$. Suppose $r'(v_n) = r(v_n)$. The transition from $\mathbf{r}$ to $\mathbf{r}'$ on the graph can be done by a sequence of transformation increasing the value of one node $u \in V - \{v_n\}$ at a time. However, after each of these transformations a walk $\mathcal{W}' = (v_0, l'_0) \rightarrow ... \rightarrow (v_n, l'_n)$ exists in the graph, as shown in Lemmas 3.3 and 3.4. Therefore, and because of Lemma 3.2, $T(\mathbf{r}') \geq T(\mathbf{r})$ which is a contradiction. □

Lemma 3.5. *If* $\mathbf{r}$ *is a retiming solution such that* $\forall v \in V : t(v, q_v) \leq T(\mathbf{r})$, *then* $\forall k \in \mathbb{Z}$ *the retiming vector* $\mathbf{r}' = [r_1 + k \cdot q_1,\ r_2 + k \cdot q_2,\ ..., r_{|V|} + k \cdot q_{|V|}]$, *is also a solution with* $T(\mathbf{r}') = T(\mathbf{r})$.

Proof. For $\mathbf{r}'$ the number of delays on edge (u, v) is

$$\begin{aligned} w'_r(u,v) &= w(u,v) + r'(v) \cdot c(u,v) - r'(u) \cdot p(u,v) \\ &= w(u,v) + (r(v) + k \cdot q_v) \cdot c(u,v) \\ &\quad -(r(u) + k \cdot q_u) \cdot p(u,v) \\ &= w(u,v) + r(v) \cdot c(u,v) - r(u) \cdot p(u,v) \\ &\quad +k \cdot (q_v \cdot c(u,v) - q_u \cdot p(u,v)) \\ &\overset{P0}{=} w(u,v) + r(v) \cdot c(u,v) - r(u) \cdot p(u,v) \\ &\overset{(3.1)}{=} w_r(u,v) \end{aligned}$$

So, number of delays on each edge are equal for $\mathbf{r}$ and $\mathbf{r}'$ and therefore $T(\mathbf{r}) = T(\mathbf{r}')$ and $\mathbf{r}'$ is a solution for all k. □

Therefore, if one retiming solution exists for T then infinite solutions exist. However, from the following equation

$$\mathbf{r}' = [r_1 + k \cdot q_1, \quad r_2 + k \cdot q_2, \ \ldots \ , \ r_{|V|} + k \cdot q_{|V|}]$$

it can be shown that $\exists k \in \mathbb{Z}$ such that

BO1 $\forall v \in V : r(v) \geq 0$

BO2 $\exists u : r(u) < q_u$

The retiming solutions for the minimum $T_{\min}$ are called optimal solutions. The solutions that satisfy BO1 and BO2 are called the basic optimal solutions. It can be proven when

the algorithm terminates it returns a basic optimal condition. The termination conditions that guarantee optimality are discussed in the next section.

3.5.2. First Termination Condition

In this section we prove the first termination condition that guarantees optimality.

Lemma 3.6. *After initialization and at each iteration of the algorithm of Figure 3.5, if* $(\exists \mathbf{r} : T(\mathbf{r}) < T_{step})$*, then the following property holds* $(\exists u : r(u) < q_u)$.

Proof. After initialization, for all nodes $r_0(v) = 0 < q_v$. If $T(\mathbf{r_0}) = T_{\min}$ then r_0 is a basic optimal solution and the lemma holds.

Suppose that $T(\mathbf{r_0}) = T_{\text{step}} > T_{\min}$. That means that there exists $\mathbf{r}$ such that $T(\mathbf{r}) < T_{\text{step}}$. Let $\mathbf{r^o}$ be a basic optimal solution with $T_{\min} = T(\mathbf{r^o}) < T_{\text{step}} = T(\mathbf{r_0})$. Initially, for all v it holds that $r_0(v) = 0 \leq r^o(v)$. Suppose that for $\mathbf{r}$ it holds that $\forall v : r(v) \leq r^o(v)$ and $T(\mathbf{r}) > T(\mathbf{r^o})$. There must exist a critical walk $\mathcal{W} = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ such that $t(v_n, l_n) + d(v_n) = T(\mathbf{r})$. By Theorem 1, we know that $r(v_n) < r^o(v_n)$. The algorithm in each iteration increases the value of v_n by 1 leaving all other r values unchanged. However, this implies that after the change $r'(v_n) = r(v_n) + 1 \leq r^o(v_n)$. And because all other r values are left unchanged, after the iteration of the algorithm $\forall v : r(v) \leq r^o(v)$. Therefore, inductively, $\forall v : r(v) \leq r^o(v)$ holds. Because of BO2, there exists $u \in V$ such that $r(u) \leq r^o(u) < q_u$. □

Lemma 3.6 also specifies a property on the existence of $T_{\min}$. In each iteration of the algorithm T_{step}, which is the minimum period found so far, is kept constant and $T(\mathbf{r})$ is

the target for reduction until $T(\mathbf{r}) < T_{\text{step}}$. Based on Lemma 3.6, if

$$\Big((\nexists u : r(u) < q_u) \Rightarrow \neg \Big(\exists \mathbf{r} : T(\mathbf{r}) < T_{\text{step}} \Big) \Big)$$

$$\Leftrightarrow \Big((\nexists u : r(u) < q_u) \Rightarrow \neg \Big(T_{\min} < T_{\text{step}} \Big) \Big)$$

$$\Leftrightarrow \Big((\forall v : r(v) \geq q_v) \Rightarrow \Big(T_{\min} = T_{\text{step}} \Big) \Big)$$

The above property makes $(\forall v, r(v) \geq q_v)$ a termination condition for the algorithm. If it is true $T_{\min} = T_{\text{step}}$ and the r-vector $(\mathbf{r} : T(\mathbf{r}) = T_{\text{step}})$, as found in the previous iterations of the algorithm is one basic optimal solution.

Theorem 3.2. *If* $(\forall v : r(v) \geq q_v)$ *the algorithm exits with one basic optimal condition.*

Proof. Follows from the discussion above. □

In the section below the second termination condition is discussed.

3.5.3. Second Termination Condition

In this section we show a second termination condition that guarantees the correctness of the algorithm.

Lemma 3.7. *After initialization and at each iteration of the algorithm of Figure 3.5, as long as* $\exists \mathbf{r}$ *such that* $T(\mathbf{r}) < T_{step}$*, for each node* v *there exists node* $u \neq v$*, such that* $(u, v) \in E$ *and* $\lfloor \frac{r(v)}{q_v} \rfloor - \lfloor \frac{r(u)}{q_u} \rfloor \leq 2$

Proof. After initialization the property holds because $\forall v : r(v) = 0$. Suppose that it holds after k iterations of the algorithm. Then at the $k+1$ iteration, let v_n be the node

with $t(v_n, q_n) + d(v_n) \geq T_{step}$. The new r value of v_n is $r'(v_n) = r(v_n) + 1$. If the critical walk of v_n is composed only of nodes v_n then $\nexists \mathbf{r}$ such that $T(\mathbf{r}) < T_{step}$ and the algorithm exits. Otherwise, even after $r(v)$ is increased there exist $u \neq v$ with $(u, v) \in E$, such that $\lfloor \frac{r(v)}{q_v} \rfloor - \lfloor \frac{r(u)}{q_u} \rfloor \leq 2$, as we show below. Moreover, if before the change there existed $z \neq v$ with $(v, z) \in E$ and $\lfloor \frac{r(z)}{q_z} \rfloor - \lfloor \frac{r(v)}{q_v} \rfloor \leq 2$, this continues to hold after the change, as well.

Let u be the last node in the critical walk with $u \neq v_n$. If u is the ith node of the walk, $v_i = u$ and $v_{i+1} = v_n$. Then in order for the walk to be valid for the successive elements (u, l_i) and (v_{i+1}, l_{i+1}), it must hold $1 \leq l_i$. A necessary condition for $1 \leq l_i$, because of Equation 3.4, is that

$$
\begin{aligned}
& l_{i+1} \cdot c(u, v_n) - w_r(u, v_n) > 0 \\
\Rightarrow \quad & l_{i+1} \cdot c(u, v_n) - w(u, v_n) - r(v_n) \cdot c(u, v_n) \\
& + r(u) \cdot p(u, v_n) > 0 \\
\overset{w(u,v_n)>0}{\Rightarrow} \quad & l_{i+1} \cdot c(u, v_n) - r(v_n) \cdot c(u, v_n) + r(u) \cdot p(u, v_n) > 0 \\
\Rightarrow \quad & l_{i+1} - r(v_n) + r(u) \cdot \frac{p(u, v_n)}{c(u, v_n)} > 0 \\
\overset{P0}{\Rightarrow} \quad & l_{i+1} - r(v_n) + r(u) \cdot \frac{q_{v_n}}{q_u} > 0 \\
\Rightarrow \quad & \frac{l_{i+1}}{q_{v_n}} - \frac{r(v_n)}{q_{v_n}} + \frac{r(u)}{q_u} > 0 \overset{l_{i+1} \leq q_{v_n}}{\Rightarrow} 1 > \frac{r(v_n)}{q_{v_n}} - \frac{r(u)}{q_u} \\
\Rightarrow \quad & 1 > \lfloor \frac{r(v_n)}{q_{v_n}} \rfloor - 1 - \lfloor \frac{r(u)}{q_u} \rfloor \Rightarrow 2 > \lfloor \frac{r(v_n)}{q_{v_n}} \rfloor - \lfloor \frac{r(u)}{q_u} \rfloor \\
\Rightarrow \quad & 2 \geq \lfloor \frac{r(v_n) + 1}{q_{v_n}} \rfloor - \lfloor \frac{r(u)}{q_u} \rfloor
\end{aligned}
$$

So, the property holds for v.

Suppose there was node z in the graph, with $(v, z) \in E$, for which $\lfloor \frac{r(z)}{q_z} \rfloor - \lfloor \frac{r(v)}{q_v} \rfloor \leq 2$. Then after the change, since $r'(v) > r(v)$,

$$\lfloor \frac{r(z)}{q_z} \rfloor - \lfloor \frac{r'(v)}{q_v} \rfloor \leq \lfloor \frac{r(z)}{q_z} \rfloor - \lfloor \frac{r(v)}{q_v} \rfloor \leq 2$$

Therefore, by induction the property holds. □

Lemma 3.8. *After initialization and at each iteration of the algorithm of Figure 3.5, if* $(\exists \mathbf{r} : T(\mathbf{r}) < T_{step})$*, then the following property holds* $(\forall v : r(v) \leq 2 \cdot q_v \cdot |V|)$.

Proof. After initialization and at each iteration of the algorithm, if $(\exists \mathbf{r} : T(\mathbf{r}) < T_{\text{step}})$, then $(\exists u : r(u) < q_u)$ (Lemma 3.6). For u the value $\lfloor \frac{r(u)}{q_u} \rfloor$ is 0.

Suppose that there exists v such that $r(v) > 2 \cdot q_v \cdot |V|$. This implies that $\lfloor \frac{r(v)}{q_v} \rfloor \geq 2 \cdot |V|$. That means that there exists $y \in V$, such that $y \neq v$ and

$$\lfloor \frac{r(v)}{q_v} \rfloor - \lfloor \frac{r(y)}{q_y} \rfloor \leq 2 \Rightarrow 2 \cdot |V| - \lfloor \frac{r(y)}{q_y} \rfloor \leq 2$$

$$\Rightarrow \quad 2 \cdot (|V| - 1) \leq \lfloor \frac{r(y)}{q_y} \rfloor \tag{3.7}$$

Continuing the same way for the rest $|V| - 2$ nodes of the graph, it can be proven that the minimum r value of the last node z of this sequence is

$$2 \cdot (|V| - (|V| - 1)) \leq \lfloor \frac{r(z)}{q_z} \rfloor$$

$$\Rightarrow 2 \leq \lfloor \frac{r(z)}{q_z} \rfloor \Rightarrow 2 \cdot q_z \leq r(z)$$

If that holds, then $(\nexists u : r(u) < q_u)$ which is a contradiction. □

From the lemma above we can conclude the correctness of the algorithm based on the termination condition $\exists v \in V : r(v) > 2 \cdot q_v \cdot |V|$.

Theorem 3.3. *If* $\exists v \in V : r(v) > 2 \cdot q_v \cdot |V|$ *the algorithm exits with one basic optimal condition.*

Proof. The reasoning for proving this theorem is the same as the one used for the proof of Theorem 3.2. □

Since the while loop of the algorithm of Figure 3.5 terminates based on the condition

$$(\forall v \in V : r(v) \geq q_v) \vee (\exists v \in V : r(v) > 2 \cdot q_v \cdot |V|)$$

the algorithm returns the basic optimal solution (Theorems 3.2, 3.3).

3.5.4. Algorithm's Complexity

From the second termination condition a bound can be derived for the number of iterations of the while loop. The sum of the r values can be

$$\sum_{v \in V} r(v) \leq \sum_{v \in V - \{u\}} (2 \cdot |V| \cdot q_v) \; + \; (q_u - 1) + 1$$

The node u is assumed to be the node that satisfies the BO2 condition. The r values of the rest of the nodes (Lemma 3.8) form the first term and 1 more move is needed to terminate the algorithm.

If as $q_{ave} = \frac{1}{|V|} \sum_{v \in V} q_v$ we represent the average q value over all nodes then the sum is upper bounded by

$$\sum_{v \in V} r(v) \leq 2 \cdot |V|^2 \cdot q_{ave}$$

Since in each iteration of the while loop the sum on the left side changes by 1, the number of iterations is bounded by $2 \cdot |V|^2 \cdot q_{ave}$.

In each iteration the necessary arrival times are computed. In the worst case the arrival computation takes

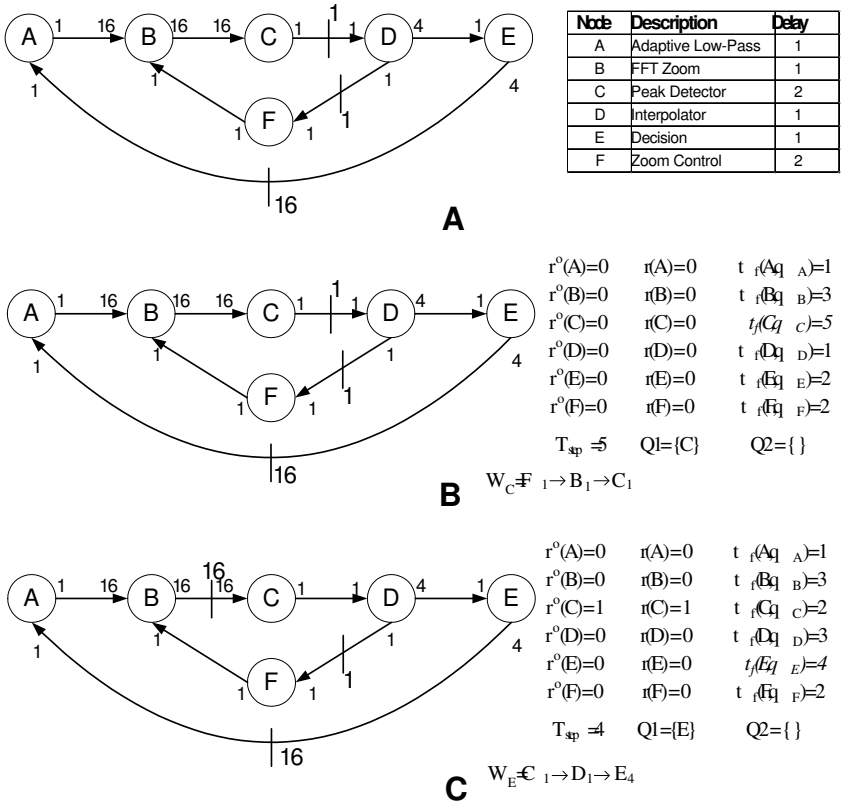
$$\sum_{\forall (u,v) \in E} q_v = |E| \cdot |V| \cdot q_{ave}$$

Therefore, the total worst case complexity is $O(|V|^3 \cdot |E| \cdot q_{ave}^2)$.

3.6. Improved Version of the Retiming Algorithm

The running time of the algorithm can be improved if we relax $P1$ not to be valid after each step of the algorithm, but be valid upon termination. That allows the algorithm to do multiple r value changes without having to find the arrival times of the node instances.

Two additional conclusions can be drawn from the previous section. First, from Theorem 3.1 we observe that the order in which we change the r values, while approaching a basic optimal solution, is not important. If there exists a critical walk in the graph and for the last node v_n of the walk $T_{step} \leq t(v_n, l_n) + d(v_n)$, then for any $\mathbf{r'}$ for which $T(\mathbf{r'}) < T_{step}$, the r value of v_n is $r(v_n) < r'(v_n)$. Second, from Lemmas 3.2-3.4 we see that by increasing the $r(v_n)$ value of a node for which $t(v_n, q_n) + d(v_n) \geq T_{step}$ cannot improve the arrival time of nodes $v_m \neq v_n$. Therefore, if before the $r(v_n)$ change, $t(v_m, q_m) + d(v_m) \geq T_{step}$ was valid, after the change $t(v_m, q_m) + d(v_m) \geq T_{step}$ remains valid.



Node	Description	Delay
A	Adaptive Low-Pass	1
B	FFT Zoom	1
C	Peak Detector	2
D	Interpolator	1
E	Decision	1
F	Zoom Control	2

Figure 3.7. The algorithm's execution on an SDF representing a spectrum analyzer (Steps A-C).

Using these conclusions, the algorithm can be modified to store all nodes, which have $t(v_m, q_m) + d(v_m) \geq T_{step}$, each time the arrival times are computed. Then modify their r values and then compute the arrival times again. That way though, it is not guaranteed that $P1$ remains invariant. Therefore, after each change all edges, which have their weight reduced, are checked for $P1$. If $P1$ does not hold, the necessary r change will be done to validate $P1$. The change is correct, as long as it is minimum, because in the basic optimal solution $P1$ must hold for all edges.

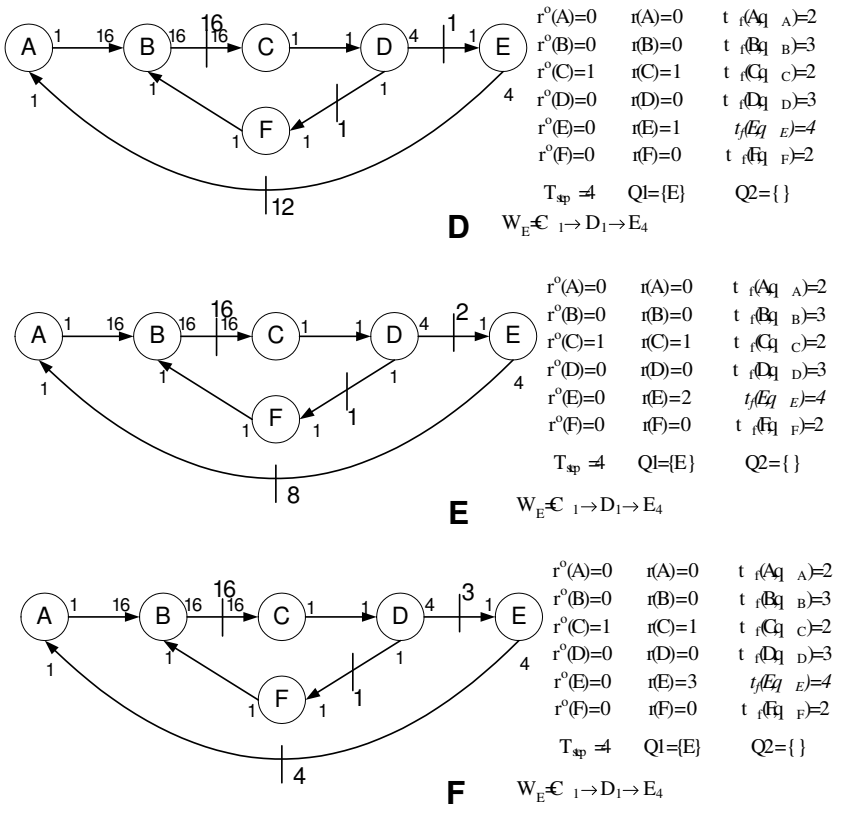


Figure 3.8. The algorithm's execution on an SDF representing a spectrum analyzer (Steps D-F).

The necessary change to make the number of delays of an edge positive is

$$w(u,v) + r(v) \cdot c(u,v) + \Delta r(v) \cdot c(u,v) - r(u) \cdot p(u,v) \geq 0$$

$$\Rightarrow \Delta r(v) \geq \lceil \frac{r(u) \cdot p(u,v) - w(u,v)}{c(u,v)} \rceil - r(v)$$

Since $r(u)$ is less than or equal to $r^o(u)$, $r(v) + \Delta r(v)$ must be less than or equal to $r^o(v)$, otherwise condition $P1$ does not hold for the basic optimal solution, which is a contradiction.

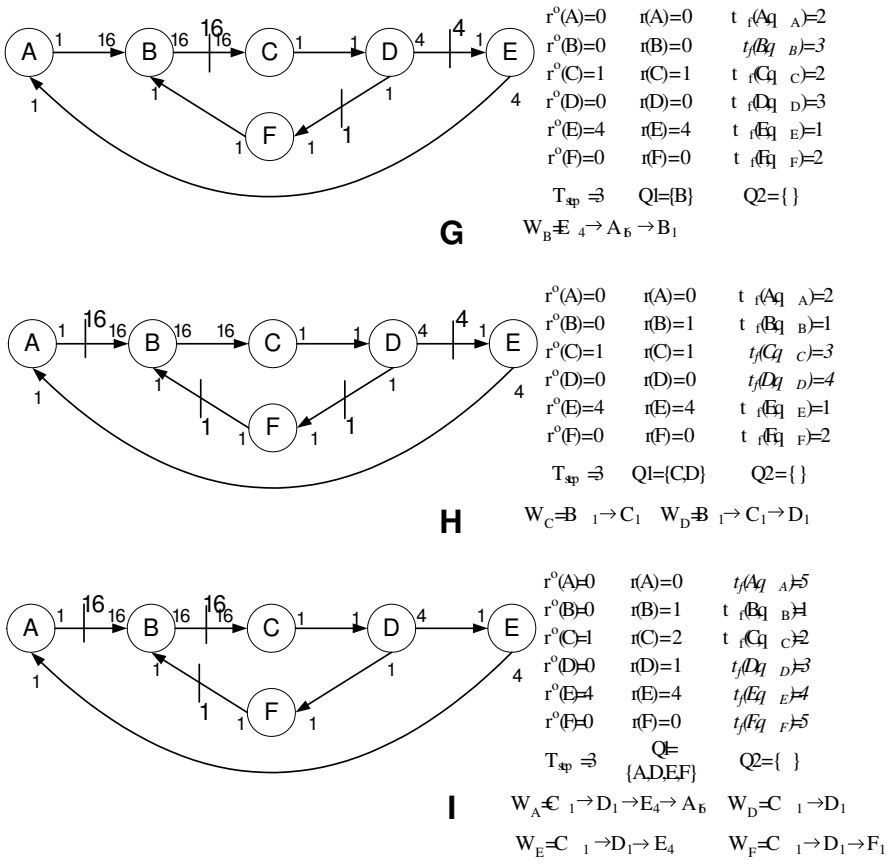


Figure 3.9. The algorithm's execution on an SDF representing a spectrum analyzer (Steps G-I).

The improved version of the algorithm can be seen in Figure 3.11. The algorithm maintains two queues. The first queue ($Q1$) holds the nodes for which it is known that their values must be increased for T_{step} to be reduced. The while loop with condition $Q1 \neq \emptyset$ increases the value of each of these nodes. The queue does not contain double entries, since when filled each node is checked only once (done by the for-loops of the algorithm).

The second queue ($Q2$) stores the edges for which $P1$ has been invalidated. For those edges, the r value of the head node is increased to restore the validity of $P1$, if needed.

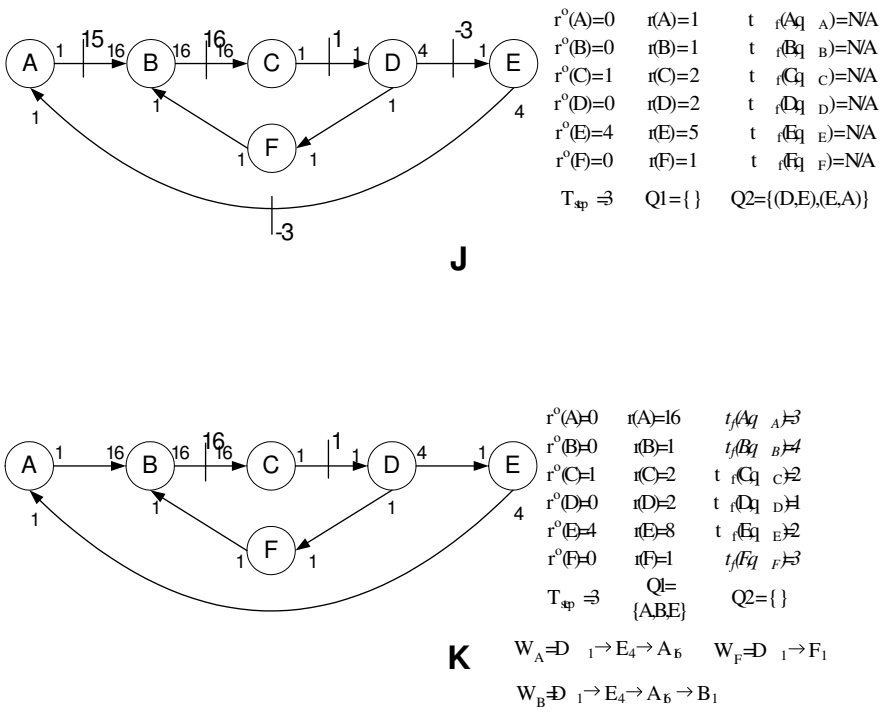


Figure 3.10. The algorithm's execution on an SDF representing a spectrum analyzer (Steps J-K).

Note that although $Q2$ does not contain double entries, the head node of two or more edges may be the same in some cases. Therefore, before restoring $P1$, it is necessary to check how large the increase of $\Delta r(u)$ should be. The check for $\Delta r(u) > 0$ in the while loop with condition $Q2 \neq \emptyset$, does exactly this.

At the end of each iteration the r values of all nodes in $Q1$ have been increased, and $P1$ has been validated for all edges, before the computation of the arrival times starts again, which generates new entries in $Q1$. In the case $maxt < T_{step}$, $Q1$'s unique entry is the node v for which $t(v_n, q_n) + d(v_n) = maxt$. Otherwise, all nodes for which $t(v, q) + d(v) \geq T_{step}$ enter the queue.

Both theorems for the termination condition are still valid.

Algorithm SDF_Retiming_Improved
Input: An SDF graph $G = (V, E, d, p, c, w)$.
Output: A pair $(\mathbf{r}, T_{min})$ which represents an optimal retiming $\mathbf{r}$ satisfying minimum complete cycle execution time T_{min}.

```
1. maxt ← 0;Q1 ← ∅;Q2 ← ∅; init_t();
2. for each v in V do
3.     r(v) ← 0;t(v, q_v) ← get_t(v,q_v);
4.     if t(v, q_v) + d(v) > maxt then maxt ← t(v, q_v) + d(v);v_n ← v;
5. endfor;
6. T_step ← maxt; Q1.enqueue(v_n);
7. while((∃v : r(v) < q_v) ∧ (∄v : r(v) > 2 · q_v · |V|)) do
8.     while(Q1 ≠ ∅)
9.       v_n ← Q1.dequeue(); r(v_n) ← r(v_n) + 1;
10.      foreach (v_n, u) ∈ E do
11.        if (w_r(v_n, u) < 0) then Q2.enqueue(v_n, u);
12.      endfor;
13.    endwhile;
14.    while(Q2 ≠ ∅)
15.      (x, u) ← Q2.dequeue();
16.      Δr(u) ← ⌈(r(x)·p(x,u) − w(x,u)) / c(x,u)⌉ − r(u);
17.      if (Δr(u) > 0) then
18.        r(u) ← Δr(u) + r(u);
19.        foreach (u, z) ∈ E do
20.          if (w_r(u, z) < 0) then Q2.enqueue(u, z);
21.        endfor;
22.      fi;
23.    endwhile;
24.    init_t();
25.    for each v in V do
26.      t(v, q_v) ← get_t(v,q_v);
27.      if t(v, q_v) + d(v) > maxt then maxt ← t(v, q_v) + d(v);v_n ← v;
28.      if t(v, q_v) + d(v) ≥ T_step then Q1.enqueue(v);
29.    endfor;
30.    if maxt < T_step then
31.      r^o ← r;T_step ← maxt;Q1.enqueue(v_n);
32.    fi;
33.endwhile;
34.Return (r^o, T_step);
```

Figure 3.11. Pseudocode describing the improved retiming algorithm.

The worst-case complexity of the algorithm remains the same. However, its practical efficiency is improved, as verified by the experimental results presented in Section 3.8.

In Figures 3.7– 3.10 the algorithm's execution on an SDF representing a spectrum analyzer is displayed:

A The spectrum analyzer graph as taken from [70].

B The values of some variables of the program when line 7 of the algorithm (Figure 3.11) is reached for the first time. The value $t_f(v, q_v) = t(v, q_v) + d(v)$ represents the finish time of q_v instance of node v. Node C has the longest finish time and is entered in Q1 to be retimed (lines 4,6).

C Values of the variables when the program reaches line 7 after the first iteration of the while loop. The minimum cycle has been reduced and the optimal solution has been updated (lines 30-31).

D-F The values of the variables for each of the next iterations of the while loop at line 7. The change of r by 1 each time guarantees the optimality of the algorithm (proof of Lemma 3.6).

G The cycle length is reduced, the optimal solution is updated, and node B is entered in Q1 (lines 30-31).

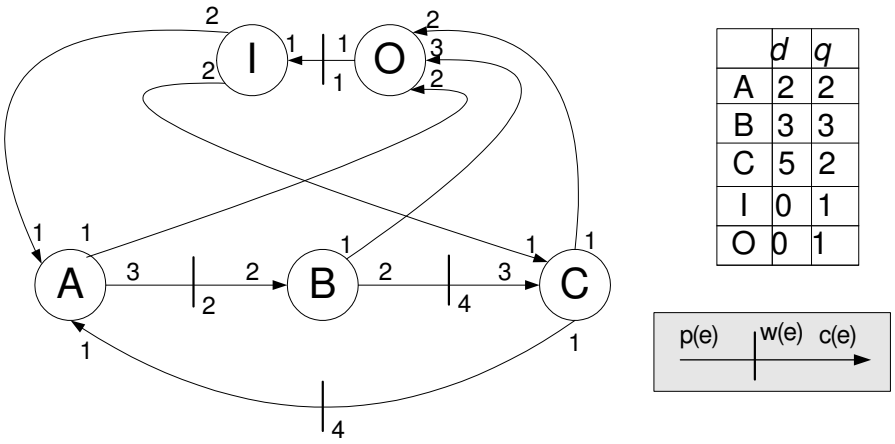
H-I Since T_{step} is not updated, all nodes with $t_f \geq T_{step}$ are entered in Q1 (line 28).

J In the next iteration some edges have negative weights. The values of the variables, when program reaches line 14 of the algorithm, are displayed. For clarity a sanitized version of the Q2 state is shown.

K After the line 14 while-loop has been executed, the r-values violate the first termination condition, therefore, the algorithm will not execute another iteration

of the line 7 while-loop. The r^o values will be returned and the retimed graph for these values is displayed in Figure G.

3.7. Source, Sink Nodes - Input Output Channels



	d	q
A	2	2
B	3	3
C	5	2
I	0	1
O	0	1

Figure 3.12. The equivalent strongly connected graph obtained by transforming the graph of Figure 3.1

The analysis presented in this chapter is based on strongly connected graphs.

If a graph has source and sink nodes, then it can be easily transformed to a strongly connected graph by introducing a new node I with $q_I = 1$ and $d(v) = 0$. Then for each source s of the graph an edge (I, s) is included in E with $c(I, s) = 1$, $p(I, s) = q_s$, and $w(I, s) = 0$. Moreover, for each sink t an edge (t, I) is included in E with $p(t, I) = 1$ and $c(t, I) = q_t$. The number of weights on these edges can be considered as a very large number W. It is easy to prove that $P0$ is still valid after this transformation and the graph is consistent.

As shown in the motivational example, there are SDF graphs which include input and output channels. These channels model the system's communication with its environment. Input and output channels are represented by edges, whose tail and head node, respectively, are missing from the graph. The head and tail of these edges are nodes that do not belong to the system under consideration, as opposed to sources and sinks of the graphs. We assume that if e is an input channel incident to node v, all $q_v \cdot c(e)$ tokens needed by v for the current complete cycle are available at time 0.

For retiming graphs with input/output channels two nodes are added I and O. All output edges are connected to O and all input edges become incident from I. The two nodes I and O are connected with an edge (O, I) with $p(O, I) = c(O, I) = w(O, I) = 1$. Moreover, $q_I = q_O = 1$ for the new nodes. Each output edge e incident from node v is replaced by (v, O) with $p(v, O) = p(e)$,$c(v, O) = c(e) \cdot q_v$, and $w(v, O) = w(e)$. In a similar way, every input edge e incident to node v is replaced by (I, v) with $c(I, v) = c(e)$,$p(I, v) = p(e) \cdot q_v$, and $w(I, v) = w(e)$. The delays of the two nodes with be $d(I) = d(O) = 0$.

These modifications on the graph have two important implications. First, since $d(I) = d(O) = 0$ the assumption that for each $v \in V$ $d(v) > 0$ does not hold anymore. This assumption was used to prove that $P1$ is an invariant (Inequality 3.5). After the addition of the new nodes the correctness of the first algorithm cannot be proven anymore. This is not a problem though for the improved version, since $P1$ is relaxed and validated again by using the $Q2$ queue. Second, the newly added edge (O, I) represents the dependence of the inputs of the next cycle on the outputs of the current cycle (Figure 3.1). Initially, $w(O, I) = 1$ and the weight of this edge should not become 0, since that would mean that the inputs for a complete cycle can be produced instantly during the complete cycle,

Graph	T			execution time (sec)		
	O'Neil's	First	Improved	O'Neil's	First	Improved
$s27$	104	104	104	0.014	0.006	0.004
$s208.1$	185	152	152	0.162	0.049	0.010
$s298$	174	174	174	0.425	0.086	0.015
$s344$	259	180	180	0.242	0.140	0.012
$s349$	310	255	255	0.693	0.153	0.024
$s382$	414	414	414	2.612	0.112	0.015
$s386$	275	275	275	0.495	0.140	0.014
$s444$	202	202	202	0.310	0.123	0.011
$s526$	632	604	604	0.859	0.314	0.061
$s641$	234	226	226	0.430	1.193	0.039
$s820$	256	247	247	1.034	0.473	0.031
$s953$	430	430	430	2.388	1.127	0.057

Table 3.2. Comparison of retiming algorithms for graphs generated with $q_{max} = 4$.

which is not a correct model of the environment of the system. In this case, the improved version of the algorithm can make

$$P6 \triangleq (w(O, I) \geq 1)$$

hold upon termination, the same way as it ensures $P1$. Edge (O, I) is entered in $Q2$ after an r change if $w(O, I) < 1$ and it is adjusted accordingly during the execution of the loop that empties $Q2$. The way this type of constraints can be handled by O'Neil's algorithm [70] is not known.

3.8. Experimental Results

In this Section we present the experimental results obtained by applying the retiming algorithms on a number of graphs. First, the experimental setup is explained. Then two

Graph	**T**			**execution time (sec)**		
	O'Neil's	First	Improved	O'Neil's	First	Improved
$s27$	129	104	104	0.084	0.005	0.006
$s208.1$	538	538	538	29.014	0.219	0.015
$s298$	765	704	704	2m:18.526	0.468	0.048
$s344$	975	905	905	6m:29.149	0.707	0.071
$s349$	1124	907	907	2m:01.058	1.187	0.108
$s382$	780	772	772	1m:04.163	1.23	0.083
$s386$	795	701	701	11.891	0.651	0.059
$s444$	1140	840	840	36.331	1.504	0.097
$s526$	1528	1498	1498	15m:05.460	2.998	0.252
$s641$	897	624	624	19.648	7.414	0.247
$s820$	895	816	816	30.478	2.548	0.140
$s953$	819	773	773	26.242	18.522	0.522

Table 3.3. Comparison of retiming algorithms for graphs generated with $q_{max} = 16$.

Graph	**T**			**execution time (sec)**		
	O'Neil's	First	Improved	O'Neil's	First	Improved
$s27$	459	416	416	1.924	0.012	0.060
$s208.1$	834	834	834	2m:50.537	1.287	0.049
$s298$	1083	1027	1027	55m:30.897	2.696	0.095
$s344$	2534	2468	2468	70m:29.472	3.457	0.415
$s349$	1503	1415	1415	8m:18.343	4.140	0.257
$s382$	1312	1273	1273	19m:29.061	5.261	0.344
$s386$	938	806	806	1m:40.775	2.733	0.129
$s444$	1185	888	888	48m:18.215	2.825	0.191
$s526$	2161	2007	2007	120m:00.000	7.796	0.479
$s641$	690	610	610	54.758	9.837	0.534
$s820$	1594	1573	1573	46m:26.437	11.805	0.622
$s953$	1776	1776	1776	5m:26.620	16.650	0.919

Table 3.4. Comparison of retiming algorithms for graphs generated with $q_{max} = 32$.

sets of experiments are presented. In the first set the graphs do not contain any zero delay nodes. In this type of graphs all algorithms are applicable. So, the three algorithms are

Graph	**T**		**Execution Time (sec)**
	Initial	Final	
$s27$	368	351	0.005
$s208.1$	1035	852	0.020
$s298$	1052	742	0.045
$s344$	1062	928	0.164
$s349$	933	833	0.016
$s382$	951	908	0.021
$s386$	745	650	0.051
$s444$	902	882	0.027
$s526$	1690	1690	0.009
$s641$	694	665	0.011
$s820$	1264	1219	0.032
$s953$	1558	1558	0.010

Table 3.5. Results for zero-delay node graphs generated with $q_{max} = 32$.

compared based on the resulting cycle length and their execution time. In the second set of experiments zero delay nodes are included in the graphs to model communication with the environment. Moreover, a constraint on the weights is applied on edge (O, I). This type of graphs can only be handled by the improved version of the retiming algorithm. The results show the algorithm can produce the optimal period even for large graphs in short execution time.

3.8.1. Experimental Setup

The graphs were obtained from the ISCAS89 benchmarks. For the delay a random integer was assigned between 1 and 30. The q value of each node was also selected randomly between 1 and the value q_{max}. Three values $(4, 16, 32)$ have been used for q_{max} to observe how the performance of the algorithms scales with this parameter. After the q value of each node was assigned, the p and c values of every edge were chosen in such a way, so that

the graph would be consistent. More specifically, $p(u,v) = \frac{q_v}{\gcd(q_u,q_v)}$ and $c(u,v) = \frac{q_u}{\gcd(q_u,q_v)}$. This method creates the minimum consumption and production rates for each edge for specific q values of the graph.

Two additional nodes I and O were included with $q_I = q_O = 1$. I was connected to all primary inputs of the graph and O to all primary outputs. Edge (O, I) was included with $w(O, I) = 1$. For graphs with no zero delay nodes $d(O)$ and $d(I)$ were chosen randomly as integers from $[1, 30]$. These values were used in the first set of experiments. In the second set $d(O) = d(I) = 0$.

Initially, non-zero weights were assigned to 50% of the total edges in the graph. The value of an edge weight was a random integer in $[1, q_{max}]$. Then the graph was checked for liveness and if a deadlock was detected, the weights of each input channel (u, v) of a node that could not execute were increased by $c(u, v)$. This process was repeated until the graph was live.

O'Neil's algorithm applies retiming to reduce the cycle length below a constraint given as an input. If the algorithm is used to find the minimum cycle length a linear search must be performed on the possible cycle length values, which are integers. Binary search cannot be performed, since it is not guaranteed that if the algorithm returns a retiming for cycle length T_1, it will not return false for cycle length $T_2 > T_1$. We implemented O'Neil's algorithm to compare it with the two new algorithms.

3.8.2. Strongly Connected Graphs with No Zero-Delay Nodes

On strongly connected graphs with no zero delay nodes all three algorithms are applicable. Tables 3.2, 3.3, and 3.4 summarize the results in terms of running time and period. The

first and improved algorithm always produce the same period T, since both of them find the optimal solution for a specific graph. The period found by these two algorithms is in all cases at least as good as the period found by O'Neil's algorithm. The difference depends on the randomly generated graph. In some cases it is 0 and in other cases it can be more then 20%. The execution time of the improved version is much faster than the other two algorithms, especially for larger graphs. As q_{ave} grows the running time of the three algorithms increases. However, the impact of that parameter is more significant for the running time of O'Neil's algorithm. The reason is that the size of the EHG and the complexity of the algorithms working on it depend on q_{ave} [35].

3.8.3. Strongly Connected Graphs with Additional Constraints

In this section the performance of the improved retiming algorithm is shown for strongly connected graphs with the additional constraint that $r(I) \geq r(O)$. This case represents the most realistic scenario for the purpose of minimizing the cycle length of SDF graphs.

Table 3.5 shows the execution time and resulting cycle length for the improved algorithm for graphs generated with $q_{max} = 32$. The other two algorithms cannot be applied on this problem instance. For O'Neil's algorithm it is not known how it can handle constraints like $P6$. Moreover, the first version of the retiming algorithm cannot handle constraints like $P6$ and under presence of zero delay nodes its correctness does not hold.

3.9. Summary

In this chapter two optimal algorithms were presented for minimum cycle length retiming of SDF graphs. The first is an optimal algorithm for retiming strongly connected

graphs, whereas the second works on any graph including graphs with input and output channels, is faster, and can handle additional constraints. The experimental results show that the improved version is orders of magnitude faster than existing approaches [70] and produces better results. In the next chapter we discuss the optimization power of synthesis operation like retiming.

CHAPTER 4

Optimization Power of Synthesis Operations

4.1. Introduction

In the previous chapter we saw how we can apply retiming to a SDF Graph. Retiming is a general synthesis operation that can be applied to structural descriptions at several abstraction levels. In this chapter we investigate the optimization power of synthesis operations like retiming. The results of this chapter can be used in two ways for system-level synthesis. First, if the nodes represent combinational modules, i.e., adders, multiplies, multiplexors, instead of gates then the power of synthesis operations at the system-level is examined. Second, even when the nodes of the structural description represent gates, RTL operations are still relevant to system-level synthesis. This is because the purpose of system-level synthesis is to generate an RTL description for the processes that will be executed in hardware. Synthesis operations, like retiming and resynthesis, can improve the efficiency of the generated RTL description.

In the rest of this chapter we use RTL representations to derive the power of the synthesis operations. However, the results hold even for higher-level representations, in which the building blocks of the structure are combinational modules instead of gates.

Logic synthesis algorithms originally targeted the optimization of PLA implementations; this was followed by research in synthesizing more general multilevel logic implementations. Currently, the central thrust in logic synthesis is sequential synthesis, i.e.,

the automatic optimization of the entire system. This is for designs specified at the structural level in the form of netlists, or at the behavioral level, i.e., in the form of finite state machines. De Micheli [26] gives an excellent introduction to logic synthesis.

We will be concerned with sequential designs. These can be specified at the behavioral level, as *finite state machines* (FSMs), or at the structural level, as *netlists* of *gates* and *registers*.

Retiming is a powerful sequential optimization step that can be applied to sequential designs described at the netlist level. It can be used to optimize the clock period or the registers area of a design. Logic synthesis is an operation that changes the circuit structure without changing the function of the combinational logic. It has been shown that given two designs, one of netlists has been derived from the other by a sequence of retiming and resynthesis, a certain equivalence relation (namely, steady-state equivalence) exists between them. However, the converse is not well understood, and there is a long history of investigations and debates on whether a sequence of retiming and resynthesis is complete for any sequentially equivalent transformation.

Malik [62] gave the first (partial) positive answer to this question. He proved that retiming and resynthesis are complete for any state re-encoding, and for some other transformations. Zhou et al. [89] provided the first negative answer by proving that some sequentially equivalent transformations cannot be done by retiming and resynthesis, which also helped to discover and fix an error in Malik's result [76]. The sweep operation, which adds or removes registers not used by any output, is needed for these transformations. However, it is an open question whether retiming and resynthesis with sweep are complete for general sequential transformations.

In this chapter, we provide a complete answer to the open question. We proved that retiming and resynthesis with sweep are complete, but with one caveat: at least one resynthesis operation needs to look through the register boundary into the logic of previous cycle. We even showed that this one-cycle reachability is required for retiming and resynthesis to be complete for re-encodings with different code length, an extension to Malik et al. [63]. It also demonstrates that reachability information cannot be captured by these structural operations. Therefore, they are complete for transformations based on all steady states unless reachability information is provided. Our completeness proof is a constructive one that applies five operations in the order of sweep, resynthesis, retiming, resynthesis, and sweep. We will discuss the implications of such a result and some practical limitations on resynthesis.

Zhou et al. [89] also started an investigation on the complexity of retiming and resynthesis verification problem. Since the general sequential equivalence verification is PSPACE-complete, a different complexity category may indicate that the gap between retiming and resynthesis and sequential transformation is big. Jiang and Brayton [39] later showed that the complexity of retiming and resynthesis verification is also PSPACE-complete. We examine their proof and point out parts that are unclear. Based on those we consider the membership of retiming and resynthesis verification an open question.

Our results have very important practical implications. Since retiming and resynthesis with sweep are complete, sequential optimization tools can be centered around them. If any reachability information is provided to the optimization, it is also critical to be supplied to the verification. Our completeness proof also indicates that the resynthesis needs to generate exponential-size circuits to complete some transformations (including

some re-encoding ones). However, no practical resynthesis is so powerful. Under realistic limitations, retiming and resynthesis verification is much simpler. Indeed, the recent sequential equivalence checking algorithms [79, 84, 40] effectively try to show that two circuits are equivalent by deriving the retiming relationships between them.

4.2. Background

Netlists and FSMs. We introduce two formalisms for representing designs, namely netlists and finite state machines (FSMs). Netlists are structural and consist of an interconnection of gates and registers. Finite state machines are behavioral and specify how the system changes its states and produces outputs responding to inputs. We leave for the readers to ponder which representation is more abstract.

Definition 4.1. *A* Finite State Machine (FSM) *is a quintuple* $(Q, I, O, \lambda, \delta)$ *where* Q *is a finite set referred to as the* states*,* I*, and* O *are finite sets referred to as the set of* inputs *and* outputs *respectively,* $\delta : Q \times I \rightarrow Q$ *is the* next-state function*, and* $\lambda : Q \times I \rightarrow O$ *is the* output function.

The output and next state functions can be inductively extended to the domains $Q \times I^+ \rightarrow O^+$ and $Q \times I^+ \rightarrow Q$, respectively; we continue to use λ and δ to denote these extended functions.

Definition 4.2. *A* netlist *is a directed graph, where the nodes correspond to elementary circuit elements, and the edges correspond to wires connecting these elements. Each node is labeled with a distinct variable* w_i*. For simplicity, we will assume that the netlist is* Boolean*, i.e. all variables take values in* $B = \{0, 1\}$*. The three basic circuit elements are* primary inputs, registers*, and* gates*. Primary input nodes have no fanins; registers*

have a single input. Associated with a gate g on n-inputs $w_1, w_2, \ldots, w_n$ *is a function from* B^n *to* B. *Some nodes are designated as being* primary outputs.

Given a value to each input and a state (an assignment of values to registers), one can uniquely compute the value of each node in the netlist by evaluating the functions at gates. A netlist η on inputs $i_1, i_2, \ldots, i_n$, outputs $o_1, o_2, \ldots, o_m$ and registers $r_1, r_2, \ldots, r_k$ bears a natural correspondence to an FSM M_η on inputs $X = B^n$, outputs $Y = B^m$, and state space $Q = B^k$. The next-state function of M_η is defined by the composed logic gates in the following manner: for each register r_i we can find a function $f_i : Q \times R_i$ by composing the functions of the gates from the inputs and register outputs to the input of the register. We will refer to f_i as the next-state function of the register i. Then $\delta_{M_\eta}(w_1, w_2, \ldots, w_n, r_1, r_2, \ldots, r_k) : Q \times X \rightarrow Q$ is simply $[f_1 f_2 \ldots f_k]$. Similarly, the output function is defined by composing the functions of gates from inputs and registers to output nodes.

Retiming, Resynthesis, and Sweep. Retiming, resynthesis, and sweep are structural operations applied on netlists.

Retiming consists of moving a given number of registers between the inputs and outputs of each combinational node. A retiming can be described mathematically by a lag function, which gives for each combinational node, the number of registers that are moved from each fanout to each fanin.

Combinational synthesis restructures the netlist within the register boundaries without changing its functionality. It leaves the FSM of the design unchanged. Retiming becomes very powerful when it is interspersed with resynthesis of the netlist within the changed

register boundaries. This is the basis for the retiming and resynthesis (RnR) paradigm proposed in [25, 63].

Sweep, the simplest among the operations, adds or removes registers not used by any output. Since synthesis normally simplifies the circuit structure, sweep is usually met as an operation removing redundant registers and logic.

Sequential Equivalence.

Definition 4.3. *Two states s and t are equivalent if and only if for every finite input sequence π, the outputs resulting on applying π are equal.*

Definition 4.4. *Two netlists C and D are FSM-equivalent if and only if every state $c \in C$ is equivalent to some state $d \in D$, and every state $d \in D$ is equivalent to some state $c \in C$.*

Definition 4.5. *The* steady state set *of a design D, denoted by D^{∞} is the subset of states such that for each state s there is an input sequence π which drives this state to itself, i.e., $\lambda_D(s, \pi) = s$. The remaining set of states is called the* transient state set.

Notice that once a design starts up in any state, it will eventually be and remain in steady states.

Theorem 4.1 ([56])**.** *If design C has been obtained from design D by a sequence of retiming moves, the steady state set of C is FSM-equivalent to the steady state set of D.*

Retiming becomes very powerful when it is combined with (combinational) resynthesis operations (the RnR paradigm). However, since resynthesis itself does not change the state transition graph of a design, we have the following corollary.

Corollary 4.1. *If design C has been obtained from design D by a sequence of retiming and combinational resynthesis moves, the steady state set of C is FSM-equivalent to the steady state set of D.*

Designated Initial State. We will not assume a designated initial state for our circuits. If we do want to force a circuits into a designated initial state we can explicitly model the reset circuitry along with the registers: indeed, this is the approach suggested for retiming initial states in [78], as opposed to the approach in [83, 32], where the implicit initial state values have to be retimed across gates.

One optimization advantage of considering designated initial states is that the synthesis algorithms have greater flexibility since the synthesis tool can potentially take advantage of don't cares arising from the set of states unreachable from the initial state. However, it is easy to show that for designs which have designated initial state, retiming and resynthesis is strictly weaker than a sequential optimization algorithm which uses unreachability don't cares (for example, [59]).

However, in general, commercial synthesis tools do not use unreachability don't cares. This is simply because computing the set of unreachable states is computationally very expensive on real designs; the theoretical complexity of this problem is PSPACE-complete:

Theorem 4.2 ([5])**.** *Given two netlists C and D, and two states s from M_C, t from M_D, checking whether s and t are equivalent is PSPACE-complete in the size of the netlists.*

4.3. Sweep is Necessary

Even though it is commonly suspected that retiming and resynthesis are not complete for all steady state transformations, Zhou et al. [89] was the first giving a proof. They designed two pairs of circuits and proved that the first pair cannot be transformed to each other even though they are FSM-equivalent. The second pair was also conjectured so. We show here that both pairs are incomplete, using the same reasoning they used for the first pair. The two pairs of circuits are shown in Figure 4.1.

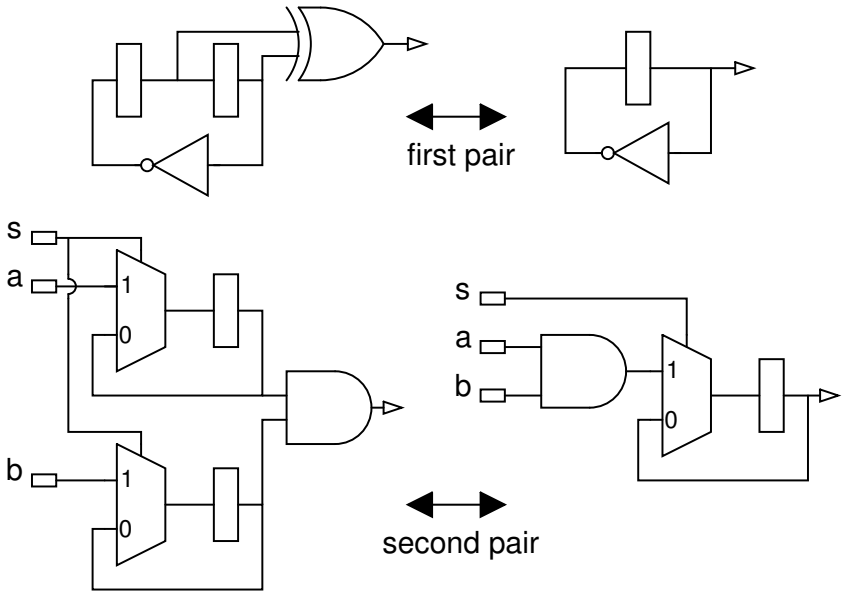


Figure 4.1. Examples showing incompleteness of retiming and resynthesis.

Lemma 4.1. *Retiming and resynthesis cannot transform one circuit to the other for either pair in Figure 4.1.*

Proof. The next state function of the left circuit in each pair contains a permutation on the set $\{0,1\}^2$, which has cardinality of 4. No matter what resynthesis does, the

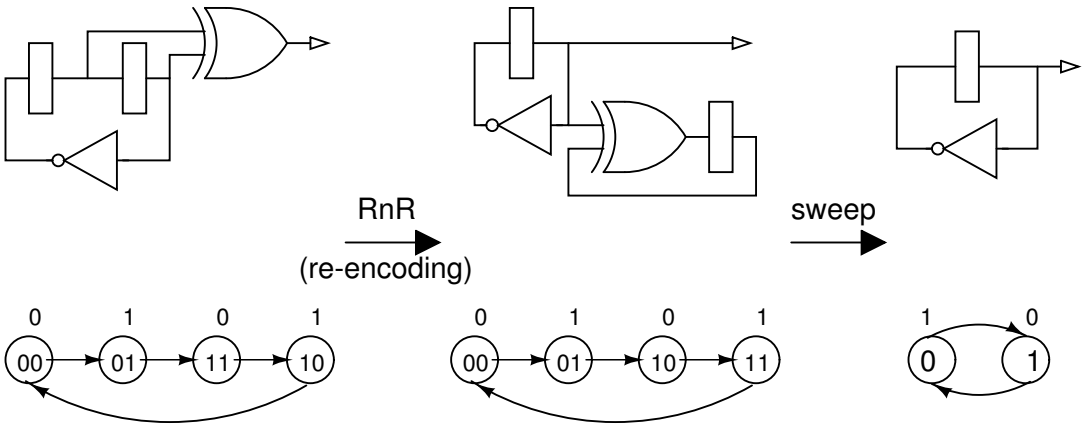


Figure 4.2. Retiming and resynthesis are more powerful with sweep.

smallest cut size on the combinational part must be at least 2. Therefore, the next retiming step cannot reduce the number of registers. Since the next state function of the new circuit still has the property as the old one, any later retiming and resynthesis steps can not reduce the number of registers, either. This means that no sequence of retiming and resynthesis can transform the left circuits to the right ones. □

However, it was also noted in [89] that with the sweep operation, the first pair of circuits are transformable to each other, as shown in Figure 4.2. We investigate whether the sweep is also of help in the second pair of circuits and find that, with re-encoding and sweep, they can be transformed, as shown in Figure 4.3. However, when trying to design a sequence of retiming and resynthesis to do the re-encoding, instead of direct applying Malik's theorem, we found that re-encoding is harder than previous thought and resynthesis needs to be slightly enhanced for retiming to be complete for re-encoding. The details will be presented in the next section.

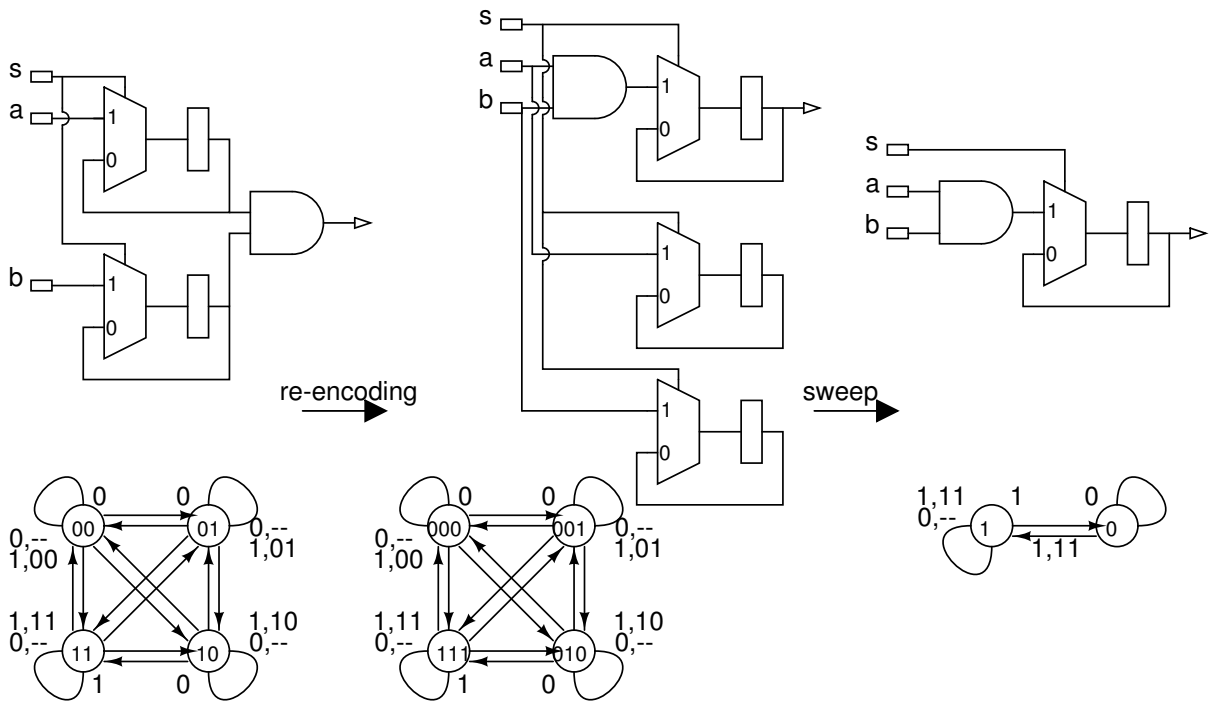


Figure 4.3. Second pair is completed by re-encoding and sweep.

4.4. Re-encoding is Hard

The first attempt to answer this question was made by Malik et al. [63] via the following result which relates designs with different state encoding:

Theorem 4.3 ([63])**.** *If two circuits have the same symbolic FSM, then one circuit can be obtained from another by a sequence of retiming and resynthesis.*

However, the above theorem cannot be applied to re-encodings with different code length. we have the following result. This result also shows a sharp difference between reachability and retiming and resynthesis.

Lemma 4.2. *Without any reachability information, some re-encodings with different code length cannot be completed by any sequence of retiming and resynthesis.*

Proof. As we already mentioned in previous section, in Figure 4.3, even though the second circuit is a re-encoding of the first one, it cannot be transformed from the first one by a sequence of retiming and resynthesis. This can be proved by considering all the states in the second circuit, including all the ignored unreachable states, as shown in Figure 4.4. Since new cycles are created in the STG of the second circuit, based on the characterization of STG transformations by Jiang and Brayton [39], retiming and resynthesis cannot produce such a circuit. □

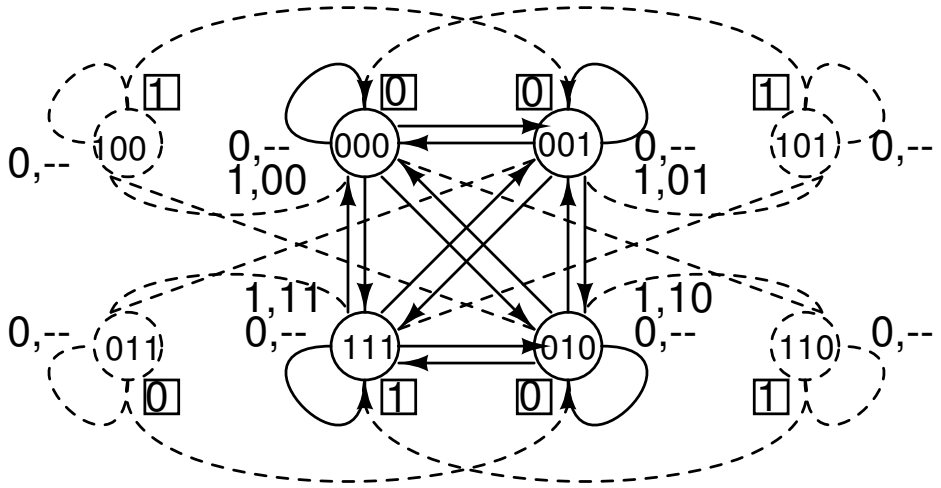


Figure 4.4. Unreachable states need to be considered in re-encoding of different length.

Although re-encodings with same code length can be done by retiming and resynthesis, their verification problem is not easy. The following theorem shows that the problem is PSPACE-hard by reducing reachability problem to it.

Theorem 4.4 ([7])**.** *Checking whether two circuits with the same number of registers are re-encoding of each other is PSPACE-hard.*

In [39] an answer about the membership of retiming and resynthesis equivalence in PSPACE is explored. Retiming and resynthesis equivalence is reduced to immediate equivalent state minimization of the two machines and then graph isomorphism starting from known initial states. It is unclear though how graph isomorphism can be checked in PSPACE. Moreover, the proof for completeness is based on the reduction of reachability to checking whether the State Transition Graphs of the two circuits are isomorphic including the transient states. The assumption is that all dangling[1] states can be merged to non-dangling states. However, due to the binary representation of the FSM, this is not always possible. An example can be seen in Figure 4.5 in which no retiming and resynthesis transformation can merge the immediate equivalent states s_1 and s_3 . The reason is that for n registers the number of states in the State Transition Graph must be 2^n. When binary representation is used and the dangling states cannot be ignored, the State Transition Graphs of two retiming and resynthesis equivalent circuits may not be transformable to isomorphic graphs.

4.5. Completeness under Reachability

We first show a revised result for re-encoding transformation.

Lemma 4.3. *When the resynthesis is allowed to use the reachability information generated from one cycle, retiming and resynthesis are complete for all re-encoding transformations, including those with different coding lengths.*

Proof. The proof is similar to Malik et al. [63], using schematics for circuits in Figure 4.6. Starting with a circuit C with the smaller encoding length n, the identity function

[1]Dangling states are inductively defined as states that have no predecessors or states whose predecessors are all dangling. All other states are considered non-dangling.

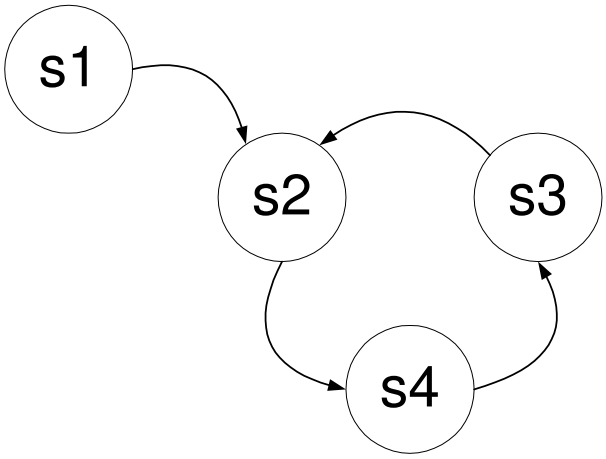


Figure 4.5. State transition graph. Immediate equivalent states s_1 and s_3 cannot be merged using retiming and resynthesis in a circuit with binary representation.

at the register outputs is resynthesized to $f \cdot f^{-1}$ where f is the one-to-one mapping from states of C to states of the target circuit D. Then retiming moves the registers forward over f. The third step resynthesizes $f^{-1} \cdot C \cdot f$ into D. However, when D is encoded on a longer length m, the one-cycle reachability information will identify the states corresponding to those in C, which will help to generate D. □

The key to the completeness of retiming and resynthesis for re-encodings is the existence of a mapping from the states of one machine to those of the other that preserves the transitions. Such a mapping is called *refinement mapping* [1].

Definition 4.6. *For two equivalent finite state machines* $(Q_1, I, O, \lambda_1, \delta_1)$ *and* $(Q_2, I, O, \lambda_2, \delta_2)$, *a* refinement mapping *is a function* $f : Q_1 \rightarrow Q_2$ *such that for any* $s \in Q_1$, s *and* $f(s)$ *are equivalent, and further for any* $i \in I$,

$$f(\delta(s, i)) = \delta(f(s), i).$$

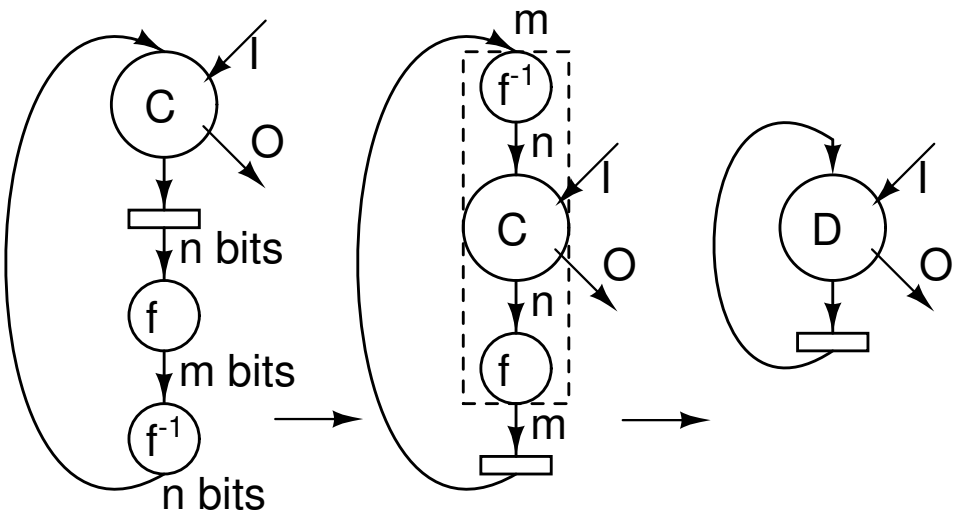


Figure 4.6. One-cycle reachability makes retiming and resynthesis complete for re-encodings.

Abadi and Lamport [1], studying the verification of one system implementing another, proved that, if S_1 implements S_2, then one can add auxiliary history and prophecy veriables to S_1 to form an equivalent system S_1^{hp} and find a refinement mapping from S_1^{hp} to S_2 under three very general hypotheses: S_1 is machine closed, S_2 has finite invisible nondeterminism, and S_2 is internally continuous. For deterministic finite state machines, they are always true. The following result is simply a corollary of the main theorem of Abadi and Lamport [1]. But we will give a direct proof to avoid detouring via general (infinite nondeterministic) system models.

Theorem 4.5. *If two deterministic FSMs C and D are equivalent, then one can add history variables to C to form an equivalent FSM C', and find an onto refinement mapping from C' to D.*

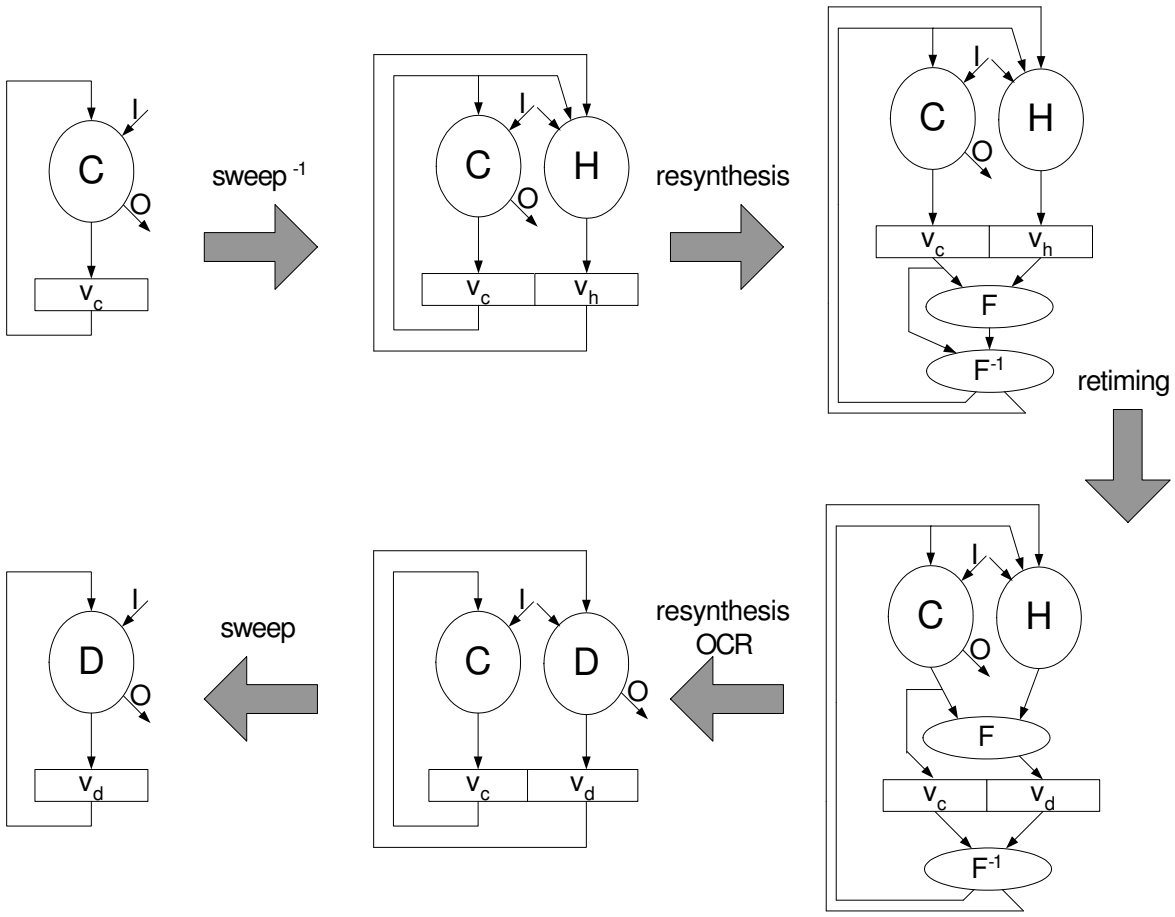


Figure 4.7. Transformation from a circuit to an equivalent one by retiming and resynthesis with sweep.

Proof. For $C = (Q_C, I, O, \lambda_C, \delta_C)$ and $D = (Q_D, I, O, \lambda_D, \delta_D)$, we can have

$$
\begin{aligned}
Q_{C'} &\triangleq \{(c,d) \in Q_C \times Q_D : c \cong d\} \\
\lambda_{C'}((c,d),i) &\triangleq \lambda_C(c,i) \\
\delta_{C'}((c,d),i) &\triangleq (\delta_C(c,i), \delta_D(d,i))
\end{aligned}
$$

It is straight-forward to check that $f(c,d) = d$ for any $(c,d) \in Q_{C'}$ is a refinement mapping from $C' \triangleq (Q_{C'}, I, O, \lambda_{C'}, \delta_{C'})$ to D. □

The proof only gives a simple construction without considering efficiency; for any states c and d such that $c \cong d$, we need only add a history variable to record the part of d that is independent of c, instead of the whole d. In the special case where each d is total dependent on c, no histroy variable is needed, and the refinement mapping is the generating function of d from c.

With the refinement mapping, a completeness result can be given as follows.

Theorem 4.6. *If two circuits are equivalent, then one of them can be transformed to the other by a sequence of sweep (inverse), resynthesis, retiming, resynthesis, and sweep, given that the second resynthesis operation is allowed to use one-cycle reachability.*

Proof. For two equivalent circuits C and D, their corresponding FSMs are determinitic and equivalent. Based on Theorem 4.5, a set of history registers and their next state functions can be added to C to make it C', and an onto refinement mapping can be found from C' to D. Denote the mapping by F. Adding unobservable registers and their next state functions is just an inverse of the sweep operation.

If F is an one-to-one mapping, then F^{-1} exists. Otherwise, we expand F with the register outputs of C and denote by F^{-1} the function that generate the state of C' from the output of F. Resynthesis can generate F and F^{-1} connected at the register output of circuit C'. Then retiming moves the registers to the outputs of F. Since F is a refinement mapping from C' to D, the relocated registers give the states of circuit D in parallel with (possibly partial) states of circuit C. The circuit composed of F^{-1}, H (the

history transition), and F can be re-synthesized into the circuit D in parallel with another circuit (partial C). Then a sweep operation will remove all unobservable part to produce circuit D. The sequence of the five operations are shown in Figure 4.7. A key operation in the second re-synthesis operation is to have the output from D, instead of C. This cannot be done if the register vectors V_c and V_d are assumed to be independent (as in pure combinational synthesis). However, with the observation that $V_d = F(V_c)$ from the previous cycle, the output O can be synthesized out solely from V_d. □

4.6. Practical Resynthesis

Retiming and resynthesis with sweep is much cheaper than doing more general sequential optimizations, e.g. extracting the set of unreachable states from an initial state and using them as don't cares [59, 18], because of the following lemma.

Lemma 4.4. *Given a circuit C and an initial state, checking whether a given state is reachable is PSPACE-complete.*

Until now we have assumed that resynthesis can make any change to the combinational part of the circuit, as long as its state transition graph is preserved. In some cases this assumption may not be realistic due to the way logic optimization algorithms work. As discussed above, some resynthesis transformations can only be performed, if a reachability relation is available. Finding such a reachability relation is computationally expensive. Moreover, resynthesis does not normally add redundant members, i.e., registers or primary inputs, in the support set of a register or primary output.

A redundant member in the support set of a function is a member whose value never matters in determining the value of the function. More formally, let r be a primary output

or a register input whose value is given by $r = f(R)$, where f is a combinational function and R is a set of register outputs and primary inputs. An element $q \in R$ is redundant if for any valuation a of $R - \{q\}$, it holds $f(a)|_{q=0} = f(a)|_{q=1}$. In such a case there always exists combinational function $\tilde{f}$, such that $\tilde{f}(R - \{q\}) = f(R)$.

The observations on the way resynthesis works help us establish structural similarities between the original and the transformed circuit. We start with property $P1$, which states that resynthesis does not increase the set of registers and inputs a register depends on. More precisely,

P1 *For any register r if $r' = f(R)$ before resynthesis and $r' = \tilde{f}(\tilde{R})$ after resynthesis, it holds*

$$\tilde{R} \subseteq R$$

where $f, \tilde{f}$ are combinational functions and $R, \tilde{R}$ sets of registers and primary inputs.

As the following lemma shows this property can be violated only if resynthesis uses reachability information or adds redundancy to the circuit.

Lemma 4.5. *If the resynthesis step does not use any reachability relation and does not add redundant members in the support set of a primary output or register, then property P1 holds.*

Proof. Assume that there exists register r in the circuit and before the resynthesis step $r = f(R)$ and after the resynthesis step $r = \tilde{f}(\tilde{R})$ and $\tilde{R} \not\subseteq R$. Then there exists register or input x such that $x \in \tilde{R}$ and $x \notin R$. Since the resynthesis step does not add redundant members in the support set, we know that the value of r cannot be a constant

and that there exist a valuation b for $\tilde{R} - \{x\}$, such that r has a different value when $x = 0$ than when $x = 1$. Without loss of generality we assume that $r = x$ for b.

Now assume that A is the set of valuations for R such that every element in $R \cup \tilde{R}$ has the same value as in b. If $R - \tilde{R} = \emptyset$, then A has only one element a for which $r = f(a)$ has a constant value and some assignment of x can violate $f(R) = \tilde{f}(\tilde{R})$. This is a contradiction as the resynthesis step guarantees that always $f(R) = \tilde{f}(\tilde{R})$ holds.

Therefore, there exists a set of registers and inputs $R_1 = R - \tilde{R}$. With the same reasoning as before we can show that there exist some valuations in A for which $r = 1$ and some for which $r = 0$. Let a be a valuation of R_1 such that $r = 0$. In order for $f(R) = \tilde{f}(\tilde{R})$ to be true, it must be the case that $a \Rightarrow (x = 0)$ holds in all reachable states. Since we assume that any combination of input values is possible at any state of the circuit, the above relation is a relation for the state variables that needs to hold in all states of the circuit. This implies that resynthesis used reachability information to exclude the states, in which $a \wedge (x = 1)$ and this is a contradiction. □

The following well-known result for retiming is useful for proving structural similarities.

Lemma 4.6. *The number of registers on a cycle is preserved during retiming.*

For all the results below the transformed circuit is the circuit obtained from the original by a sequence of retiming and resynthesis transformations. Let $N(C)$ denote the number of registers on cycle C of the graph.

Lemma 4.7. *Each cycle of the transformed circuit can be mapped to a cycle of the original circuit with the same number of registers.*

Proof. Assume we have a circuit $G_1 = (V_1, E_1)$ and we apply a countable sequence of retiming and resynthesis transformations to obtain circuit $G_n = (V_n, E_n)$. We prove the lemma by induction. For the base case the graph G_1 is isomorphic to itself and, therefore, the mapping is from each cycle to itself. Now assume that for every cycle C_k of graph G_k obtained after k steps of retiming and resynthesis transformations there exists a cycle C_1 in G_1 with the same number of registers.

We want to prove that for every cycle C_{k+1} of G_{k+1} there exists a cycle C_1 in G_1 with $N(C_{k+1}) = N(C_1)$. Using the induction hypothesis, it is sufficient to show that every cycle of G_{k+1} can be mapped to a cycle of G_k with the same number of registers.

In case G_{k+1} is obtained from G_k by retiming, the structure of the graph remains the same. The reason is that no vertices or edges are added or removed and only the numbers of registers on the edges change. From Lemma 4.6 we know that the number of registers on each cycle are preserved. Consequently, each cycle of G_{k+1} can be mapped to the cycle with the same sequence of edges in G_k, which has the same number of registers.

For the case G_{k+1} is obtained from G_k by resynthesis we assume C_{k+1} is any cycle in G_{k+1}. Since combinational cycles are not allowed, cycle C_{k+1} must have at least one register. That means there exists $m \in \mathbb{N}$ and a sequence of registers $r_1, ..., r_m$ with a cyclic dependency, i.e.,

$$\begin{aligned} r_m = \tilde{f}_m(\tilde{R}_{m-1}) \quad &\wedge \quad r_{m-1} \in \tilde{R}_{m-1} \\ r_{m-1} = \tilde{f}_{m-1}(\tilde{R}_{m-2}) \quad &\wedge \quad r_{m-2} \in \tilde{R}_{m-2} \\ \vdots \qquad &\vdots \\ r_1 = \tilde{f}_1(\tilde{R}_m) \quad &\wedge \quad r_m \in \tilde{R}_m \end{aligned}$$

Resynthesis does not remove registers from the circuit. Moreover, because of Property P1, we have that for all $j \in 1..m : r_{j\oplus_m 1} = f_{j\oplus_m 1}(R_j) \Rightarrow \tilde{R}_j \subseteq R_j$[2]. This implies that $r_j \in R_j$, and, therefore, the cyclic dependency exists in G_k, as well. In addition, the specific cycle in G_k has the same number of registers, i.e., m, as C_{k+1}. □

Using the Lemma above we can prove the following lemma for paths from primary inputs to cycles of the graph.

Lemma 4.8. *Each path from an input pin to a cycle with n_r registers of the transformed circuit can be mapped to a path from the same input pin to a cycle with n_r registers of the original circuit.*

Proof. As in the case of Lemma 4.7 the proof is by induction. Circuit $G_1 = (V_1, E_1)$ is the initial circuit to which we apply a sequence of retiming and resynthesis transformations to obtain circuit $G_n = (V_n, E_n)$. The base case is trivial, since G_1 is isomorphic to itself.

Now assume that graph G_k is obtained after k steps of retiming and resynthesis transformations. For every path P_k from a primary input i to a cycle with n_r registers of graph G_k there exists a path P_1 in G_1 that connects the same primary input i to a cycle with n_r registers.

It is sufficient to show that if G_{k+1} is obtained from G_k by retiming or resynthesis, then for every path P_{k+1} of G_{k+1} from primary input i to a cycle with n_r registers can be mapped to a path P_k of G_k from the same primary input i to a cycle with the same number of registers. Then because of the induction hypothesis by composition of the mappings the paths of G_{k+1} can be mapped to the paths of G_1.

[2]The expression $a \oplus_m b$ denotes $a + b$ modulo m.

In case G_{k+1} is obtained from G_k by retiming the two graphs are isomorphic to each other. Then each path P_{k+1} can be mapped to the path P_k with the same sequence of edges. Because of Lemma 4.6 the two paths connect to cycles with the same number of registers.

In case G_{k+1} is obtained from G_k by resynthesis let P_{k+1} be any path from a primary input i to a register r in cycle C_{k+1}. From Lemma 4.7 we know that there exists cycle C_k in G_k with the same number of registers as C_{k+1} that includes register r. Consequently, it is sufficient to show that there exists path P_k from i to r in G_k.

From the existence of P_{k+1} we know that there exists $m \in \mathbb{N}$ such that

$$\begin{aligned} r_m = \tilde{f}_m(\tilde{R}_{m-1}) \quad &\wedge \quad r_{m-1} \in \tilde{R}_{m-1} \\ r_{m-1} = \tilde{f}_{m-1}(\tilde{R}_{m-2}) \quad &\wedge \quad r_{m-2} \in \tilde{R}_{m-2} \\ \vdots \qquad &\vdots \\ r_1 = \tilde{f}_1(\tilde{R}_0) \quad &\wedge \quad i \in \tilde{R}_0 \end{aligned}$$

with $r = r_m$. Since for all $j \in 0..m-1 : \tilde{R}_j \subseteq R$, a path from i to r exists in G_k, as well. □

Similarly, we can prove the following result for paths from cycles to the outputs

Lemma 4.9. *Each path from a cycle with n_r registers to an output pin of the transformed circuit can be mapped to a path from a cycle with n_r registers to the same output pin of the original circuit.*

Finally, the following lemma can be proved for input to output paths.

Lemma 4.10. *Each path from a primary input to a primary output in the transformed circuit can be mapped to a path connecting the same primary input to the same primary output in the original circuit.*

Procedures for sequential equivalence checking exploit structural similarities to simplify the verification problem [84, 40]. We believe that the above results will have practical implications and help researchers design more efficient verification algorithms.

4.7. Summary

We have shown that retiming and resynthesis with sweep are almost complete for all steady state equivalent transformations, in the sense that resynthesis needs to get one-cycle reachability information by looking into previous phase. Without such information, they cannot even complete re-encodings with different code length. It suggests that a powerful sequential optimization tool can be built around retiming, resynthesis, and sweep, and also suggests to enhance each resynthesis step to employ one-cycle reachability by looking into previous phase. In practice, resynthesis may not generate exponential-size circuits and may have other restrictions. Those restrictions can make the retiming and resynthesis equivalence checking easier. In the next chapter we will consider the combined synthesis-verification problem.

CHAPTER 5

Combined Synthesis and Verification

5.1. Introduction

To cope with the increasing complexity of digital circuits, design methodologies move to higher-levels of abstraction. The push to higher abstraction levels provides synthesis with more opportunities for optimization, but makes the verification task more complex. Synthesis algorithms that optimize sequential circuits, i.e., circuits containing registers besides combinational logic, are not an exception.

As we saw in Chapter 4, a powerful optimization technique for sequential circuits is a sequence of retiming and resynthesis operations [63] (*RnR sequence*). Resynthesis is a combinational transformation that can be applied to blocks of logic between registers. Retiming is a sequential transformation that moves registers across gates generating new logic blocks that give resynthesis new opportunities for optimization. The optimization power of the RnR sequence has been discussed in many works [89]. Despite its optimization power, the RnR sequence is not widely used due to the complexity of checking sequential equivalence [39] between the initial and final design. There is a need, therefore, for efficient verification methods that preserve the optimization power of the retiming and resynthesis sequence.

Van Eijk developed an efficient method for checking sequential equivalence between two designs that is based on finding equivalent signals in the two circuits [85]. Jiang

et. al. showed that the method is complete for sequences of retiming and resynthesis transformations with no more than one resynthesis step [40]. If more than one resynthesis step is applied and the verification procedure shows that the outputs are not equivalent, no conclusion can be drawn.

Ashar et. al. demonstrated that circuits with the Complete-1-Distinguishability (C-1-D) property can be verified with an efficient and complete method [4]. In C-1-D circuits each pair of distinct states produces different output values for some input and, therefore, each state is distinguishable from any other in a single cycle. If one of the two circuits to be checked for equivalence satisfies the C-1-D property, sequential equivalence checking can be reduced to combinational equivalence checking. Not all circuits satisfy C-1-D and, therefore, the authors developed a method to enforce this property by modifying the structure of the circuit. However, a side effect of the modifications to enforce C-1-D is that the optimization power of retiming and resynthesis is reduced.

A complete method to check for sequential equivalence of two circuits without restrictions on the synthesis part is reachability analysis [23]. Starting with the initial state of the circuits a forward traversal of the state space can be performed to check whether a "bad state", i.e., a state that shows the two circuits are not equivalent, can be reached. During each iteration the method uses the next state relation to increase the set of reachable states. In backward reachability analysis the process starts from the set of bad states and checks whether an initial state is reachable using the inverse of the next state relation. The number of iterations that this method requires to produce a useful answer is generally hard to compute. Without this bound, if the set of reachable states does not converge

after a specific number of iterations, no conclusion can be drawn for the correctness of the transformations.

To improve the efficiency of reachability analysis and reduce the number of iterations without destroying completeness, a number of structural optimizations have been proposed [50, 36]. For example, retiming can be used to reduce the number of variables before the traversal starts. These techniques can be used in conjunction with the ideas proposed in this chapter.

The approach we describe targets the equivalence checking of a pair of circuits, one of which has been obtained from the other by a sequence of retiming and resynthesis transformations. We extend the C-1-D property to C-k-D, where $k \in \mathbb{N}$. A circuit fulfills the C-k-D property if every two non-equivalent states can be distinguished in k cycles or less. The contributions of our work are the following:

(1) We extend C-1-D to C-k-D for an integer k. Since every circuit with the C-k-D property satisfies also the C-m-D for all $m \geq k$, our approach is more general than C-1-D. Therefore, it can be applied to more circuits, if we are not allowed to modify the circuit before synthesis.

(2) We show how we can prove sequential equivalence for circuits that satisfy the C-k-D property by unrolling the product circuit a bounded number of times. The presented verification technique is complete, in the sense that if it fails we know that there is a problem with the retiming and resynthesis transformations we applied.

(3) We present a method to modify a circuit, so that it satisfies C-k-D.

(4) We derive a method to apply a sequence of retiming and resynthesis transformations to the modified circuit, so that the optimization power of retiming and resynthesis is not restricted. This implies that the obtained optimized circuit is the same as the circuit obtained without the modifications we applied to make its verification easier. Moreover, the complexity of retiming and resynthesis is not increased by the modification.

Our experimental results show that enforcing the C-k-D property with additional logic and outputs can speed up the verification procedure.

The modification on the circuit to enforce the C-k-D property, i.e., contribution 3, is similar to target enlargement [10]. However, the method we propose is applicable even when BDD construction cannot be completed. Moreover, it is a structural transformation that is applied before synthesis and without affecting the synthesis result it improves the verification running-time.

In the next section we describe the notation we use and we define the problem we want to solve. Then in Section 5.3 we define the C-k-D property and show how it is related to output equivalence. Moreover, we describe a method to modify a circuit to satisfy the C-k-D property. In Section 5.4 we explain how we can check sequential equivalence for a pair of circuits, one of which satisfies the C-k-D property and the other is obtained from the first by a sequence of retiming and resynthesis transformations. Then in Section 5.5 we demonstrate a method to apply unrestricted retiming and resynthesis to a circuit which has been modified to satisfy the C-k-D property. The method is independent of the way C-k-D was enforced. Finally, in Section 5.6 our experimental results are described and in Section 5.7 we share our conclusions.

5.2. Preliminaries

In this section we introduce the concepts we use in this chapter. Some of them have already been defined in Section 4.2, but we repeat them here for the reader's convenience.

5.2.1. Models

We introduce two formalisms for representing circuits, namely netlists and finite state machines (FSMs). Netlists are structural and consist of an interconnection of gates and registers. Finite state machines are behavioral and specify how the system changes its states and produces outputs responding to inputs.

A *netlist* is a directed graph, where the nodes correspond to elementary circuit elements, and the edges correspond to wires connecting these elements. The three basic circuit elements are *primary inputs, registers*, and *gates*. Primary input nodes have no fanins; registers have a single input. Associated with a gate g on n-inputs $w_1, w_2, \ldots, w_n$ is a function from B^n to B, where $B = \{0, 1\}$. Some nodes are designated as being *primary outputs*.

A *Finite State Machine (FSM)* is a quintuple $(\Sigma, I, O, \lambda, \delta)$ where Σ is a finite set referred to as the *states*, I, and O are finite sets referred to as the set of *inputs* and *outputs* respectively, $\delta : \Sigma \times I \rightarrow \Sigma$ is the *next-state function*, and $\lambda : \Sigma \times I \rightarrow O$ is the *output function.*

The output and next state functions can be inductively extended to the domains $\Sigma \times I^+ \rightarrow O^+$ and $\Sigma \times I^+ \rightarrow \Sigma$, respectively, where A^+ is the set of finite, non-empty sequences of elements of the set A. We continue to use λ and δ to denote these extended functions.

Given a value to each input and a state (an assignment of values to registers), one can uniquely compute the value of each node in the netlist by evaluating the functions at gates. A netlist η on inputs $i_1, i_2, \ldots, i_n$, outputs $o_1, o_2, \ldots, o_m$ and registers $r_1, r_2, \ldots, r_k$ bears a natural correspondence to an FSM M_η on inputs $X = B^n$, outputs $Y = B^m$, and state space $\Sigma = B^k$. The next-state function of M_η is defined by the composed logic gates in the following manner: for each register r_i we can find a function $\delta_i : \Sigma \times X$ by composing the functions of the gates from the inputs and register outputs to the input of the register. We will refer to δ_i as the next-state function of the register i. Then $\delta_{M_\eta}(w_1, w_2, \ldots, w_n, r_1, r_2, \ldots, r_k) : \Sigma \times X \rightarrow \Sigma$ is simply $[\delta_1 \delta_2 \ldots \delta_k]$. Similarly, the output function is defined by composing the functions of gates from inputs and registers to output nodes.

Two circuits are called *compatible* if they have the same set of primary inputs and primary outputs. We restrict the equivalence problem on pairs of compatible circuits. For two compatible circuits C_a and C_b the product circuit $C_x = C_a \times C_b$ is defined. The netlist of the product circuit is built by joining the corresponding primary inputs and connecting the corresponding outputs to xor-gates. The outputs of those xor gates become the outputs of the product circuit. The outputs of the product circuit are all zero in state s, if the outputs of the two circuits are equal in s. The set Σ_x of states of the product circuit is given by $\Sigma_a \times \Sigma_b$, where Σ_a, Σ_b are the set of states of C_a and C_b.

From the FSM the *State Transition Graph* (STG) of the circuit can be built. The STG has one node for each state and its edges represent the transitions that can occur between states in one clock cycle. The longest shortest path between any two nodes of the STG is called the *diameter* of the circuit.

For any circuit element o, we denote as $o^{(x)}$ the value of the element after x cycles. For a predicate ϕ over circuit elements (e.g., registers, inputs), $\phi^{(x)}$ is obtained by replacing each circuit element with its value after x cycles.

5.2.2. Retiming and Resynthesis

Retiming and resynthesis are structural operations applied on netlists.

Retiming consists of moving a given number of registers between the inputs and outputs of each combinational node [57]. A retiming can be described mathematically by a lag function, which gives for each combinational node, the number of registers that are moved from each fanout to each fanin.

Resynthesis restructures the netlist within the register boundaries without changing its functionality. It leaves the FSM of the design unchanged.

Retiming becomes very powerful when it is interspersed with resynthesis of the netlist within the changed register boundaries. This is the basis for the retiming and resynthesis (RnR) paradigm proposed in [63].

5.2.3. State Characterization and Equivalence

We assume an initial state is specified for each circuit. After starting from the initial state, the set of states that the circuit can enter are called *reachable*.

A state s of a circuit is *dangling* , iff either it has no predecessor states or all its predecessor states are dangling [39]. Let D represent the set of dangling states. The states in $\Sigma - D$ are called *non-dangling* (Figure 5.1).

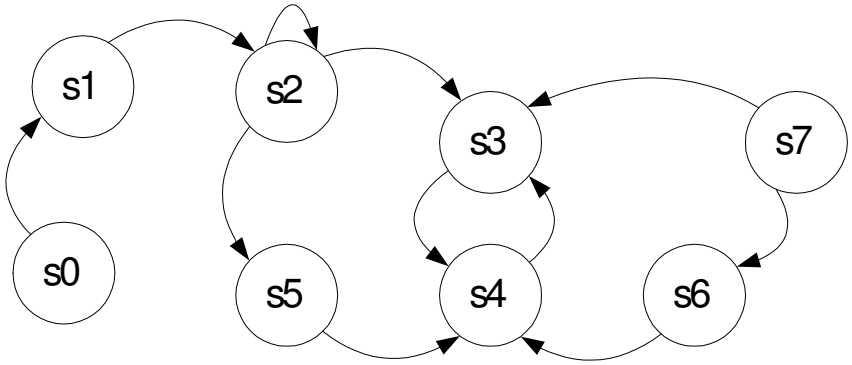


Figure 5.1. In the displayed STG states $s0$, $s1$, $s6$, and $s7$ are dangling. The rest are non-dangling states.

Let C_a and C_b be two compatible circuits with FSMs $(\Sigma_a, I, O, \lambda_a, \delta_a)$ and $(\Sigma_b, I, O, \lambda_b, \delta_b)$. Then two states $s_a \in \Sigma_a$ and $s_b \in \Sigma_b$ are *equivalent*, i.e., $s_a \approx s_b$, if and only if

$$\forall i \in I^+ : \lambda_a(s_a, i) = \lambda_b(s_b, i)$$

Two compatible circuits with a prespecified initial state are *sequentially equivalent*, iff their initial states are equivalent. Two compatible circuits are *RnR equivalent* , iff one can be obtained from the other by a sequence of retiming and resynthesis transformations.

5.2.4. Complete-1-Distinguishability

Definition 5.1. *A circuit C satisfies the C-1-D property iff for every pair of non-equivalent states s_1 and s_2 there exists some input value $i \in I$ such that $\lambda(s_1, i) \neq \lambda(s_2, i)$.*

Note that our definition of C-1-D property is more general than previous approaches [4] because we require that non-equivalent states are distinguishable, as opposed to distinct

states. Clearly, every circuit that has the C-1-D property, as defined in [4], satisfies the C-1-D property defined above.

5.2.5. Problem Formulation

Assume we have two circuits C_a and C_b. Circuit C_b has been obtained from C_a by a sequence of retiming and resynthesis transformations. The purpose of those transformations is to create an optimized and sequentially equivalent version of C_a.

Definition 5.2. *Complete RnR Equivalence Checking Problem: Prove that C_a and C_b are sequentially equivalent or find whether C_b cannot be obtained from C_a by a sequence of retiming and resynthesis operations.*

We will refer to the Complete RnR Equivalence Checking Problem as RnR checking for simplicity. Obviously, this problem can be solved by model checking. However, we are interested in forcing a bound to the number of iterations after which we give a solution to the RnR checking problem. This bound should be forced without restricting the optimization power of the synthesis part, i.e., the retiming and resynthesis sequence.

5.3. Exploiting the Output Equivalence to Derive C-k-D

We are given two compatible circuits; the original circuit C_a and the transformed circuit C_b. Our first step for proving their equivalence is to build the product circuit C_x. In this section we show how to exploit the structure of C_x starting from the outputs to derive a relation between the registers of C_a and C_b (Section 5.3.1). Then we define C-k-D and show how it is related to the characteristic predicate of the relation between

the registers.(Section 5.3.2). This predicate is used to derive an invariant for C_x. By modifying the structure of the circuit, we can enforce C-k-D (Section 5.3.3).

5.3.1. Output Equivalence

Every output of the product circuit C_x has the value 0 in all reachable states, if and only if C_a and C_b are sequentially equivalent. By construction of C_x this is equivalent to corresponding outputs o_a and o_b being equal in all those states. Outputs o_a and o_b can be expressed as a function of the registers and inputs that drive them. The function can be extracted by traversing the netlist backwards from the outputs until a register or a primary input is met. Since the inputs cannot be restricted, this is a relation over the registers of the two circuits that must hold for all input values. More precisely, let λ_a, λ_b be the combinational functions that give the values of o_a and o_b, and $R_a = \{p_1, p_2, ..., p_m\}$, $R_b = \{q_1, q_2, ..., q_n\}$, I_a, I_b be the set of registers and inputs that are connected to o_a and o_b by a combinational path, then

$$o_a = o_b \quad \Leftrightarrow \quad \forall i \in I : \lambda_a(p_1, p_2, ..., p_m, i) = \lambda_b(q_1, q_2, ..., q_n, i) \tag{5.1}$$

where $I = I_a \cup I_b$.

We define as χ_0 the characteristic predicate over the registers in $R_a \cup R_b$ that are connected to the outputs by a path of 0 registers, i.e.,

$$\chi_0 \quad \triangleq \quad \forall i \in I : \lambda_a(p_1, p_2, ..., p_m, i) = \lambda_b(q_1, q_2, ..., q_n, i)$$

Predicate χ_0 is satisfied by those states of the product circuit that have an output value of 0 for any input.

As we saw in Section 5.2.1, the next state function δ is composed of a number of δ_i functions, each giving the value of a register in the next state as a function of the registers and the input. Function δ_{r_i} of register r_i can be extracted from the circuit by traversing the signal that drives r_i backwards until a register or an input is met. By replacing the registers of (5.1) with the functions giving their value in the next state, predicate χ_1 is generated.

Predicate χ_1 defines a relation between the register values of the two circuits that are connected by a path of exactly one register to the outputs o_a and o_b. The predicate uses the inputs connected to those outputs by a path of 0 or 1 register to determine which states of the product circuit cause $o_a = o_b$ in the next cycle for any input, i.e.,

$$\chi_1 \Rightarrow o_a^{(1)} = o_b^{(1)}$$

or, equivalently,

$$\chi_1 \Rightarrow \chi_0^{(1)}$$

As an example, consider the two circuits shown in Figure 5.2. For those circuits we have

$$\begin{aligned} \chi_0 &= (p1 \vee p2) \leftrightarrow (q1 \wedge q2) \\ \chi_1 &= \forall i \in \{0,1\} \,:\, ((i \wedge p4) \vee p3) \leftrightarrow \big((i \wedge q_4) \vee \overline{q3}\big) \end{aligned}$$

States $s_a = \{p1 = 0, p2 = 0, p3 = 0, p4 = 1\}$ and $s_b = \{q1 = 1, q2 = 0, q3 = 1, q4 = 0\}$ satisfy χ_0. Therefore, the output of the product circuit is zero in state (s_a, s_b). However,

(s_a, s_b) does not satisfy χ_1. For input $i = 1$ the next state of the product circuit violates χ_0 and the output is 1.

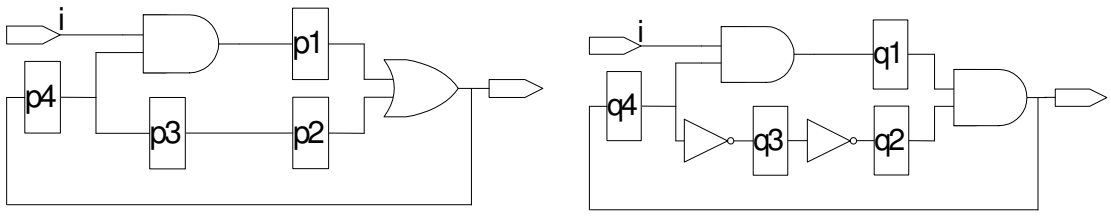


Figure 5.2. Two example circuits.

Similarly, by processing the circuit structure we extract χ_k from χ_{k-1} for any $k \in \mathbb{N}$. Moreover, we build the predicates in the same way for all outputs and for each $k \geq 0$ we take their conjunction. Therefore, the states that satisfy χ_k guarantee that for all xor-ed output pairs (o_a, o_b) and for all input values, $o_a = o_b$ after k cycles. By construction, it holds

$$\chi_k \Rightarrow \chi_{k-1}^{(1)} \tag{5.2}$$

for all $k \geq 1$. By the definition of χ_k, for a pair of states satisfying $\neg\chi_k$ there exists an input sequence of length k, such that a pair of corresponding outputs is different after k cycles.

States of the product circuit that satisfy the conjunction

$$\psi_k \triangleq \bigwedge_{i \in 0..k} \chi_i$$

have the same output values for the next k cycles. States of the two circuits that are equivalent must satisfy ψ_k for any k, i.e.,

$$\forall(s_a, s_b) \in \Sigma_a \times \Sigma_b \ : \ s_a \approx s_b \ \Rightarrow \ (s_a, s_b) \models \psi_k \tag{5.3}$$

In the worst case the evaluation of ψ_k for a state requires I^{k+1} input values. However, by the method described above only the inputs relevant to producing the output values are processed in each of the $k+1$ cycles.

5.3.2. Complete-k-Distinguishability

In this section we define the C-k-D property and show how we can find whether a circuit satisfies it. First, we define a property for states.

Definition 5.3. *A pair of states s_1 and s_2 of circuit C is k-Distinguishable iff there exists some input sequence i of length $m \leq k$, such that $\lambda(s_1, i) \neq \lambda(s_2, i)$.*

Definition 5.4. *A circuit C satisfies the C-k-D property, iff every pair of non-equivalent states s_1 and s_2 is k-Distinguishable.*

If we take the product C_x of C with itself, then states s_1 and s_2 are k- Distinguishable, if and only if

$$(s_1, s_2) \not\models \psi_{k-1}$$

Therefore, circuit C has the C-k-D property, if and only if

$$\forall(s_1, s_2) \in \Sigma_x \ : \ s_1 \approx s_2 \ \Leftrightarrow \ (s_1, s_2) \models \psi_{k-1} \tag{5.4}$$

which follows from the definition of C-k-D and (5.3).

By the definition of C-k-D, if any circuit satisfies the C-k-D property, then it satisfies the C-m-D property for all $m \geq k$. Therefore, C-1-D is the most restricted property of this class. More specifically, the circuits satisfying C-1-D are a subset of the circuits satisfying C-k-D for any $k \geq 1$.

Lemma 5.1. *For every circuit C there exists $k \in \mathbb{N}$ such that C has the C-k-D property, where k is bounded from above by the diameter of $C_x = C \times C$.*

5.3.3. Convergence

For verification purposes it is useful to have an invariant of the product circuit that implies correctness. Property ψ_{k-1} implies output equivalence in the current state by construction. In this section we show that for a circuit C that satisfies C-k-D, ψ_{k-1} is also an invariant of the product of C with itself (Lemma 5.2). Moreover, we show how we can transform the product circuit $C_x = C \times C$, so that formula (5.4) holds for C_x. Then in Section 5.4 we show how we can use ψ_{k-1} to check equivalence between the original and the transformed circuit.

Lemma 5.2. *If C fulfills the C-k-D property, then ψ_{k-1} is an invariant of C_x.*

Proof. From (5.4) we know that $(s_1, s_2) \models \psi_{k-1}$ if and only if s_1 and s_2 are equivalent. The equivalence of these states implies that for any input $i \in I$ their next states are also equivalent, and, therefore, satisfy ψ_{k-1} (Formula 5.3). □

As we can see from the proof of Lemma 5.2, it is sufficient that C_x satisfies formula (5.4) for ψ_{k-1} to be an invariant of C_x.

In Figure 5.3 a Venn diagram of the state space of the product circuit can be seen. The sets of states satisfying ψ_{k-1}, .., ψ_0 are displayed. The set of states satisfying ψ_m is a subset of the states satisfying ψ_{m-1}, as $\psi_m = \psi_{m-1} \wedge \chi_m$. In order for ψ_{k-1} not to be an invariant, there must be a state s satisfying ψ_{k-1} and having a next state s_1, such that $s_1 \not\models \psi_{k-1}$. That means that $s_1 \models \neg\chi_{k-1}$, since $s \models \bigwedge_{i\in 0..k-1} \chi_i$ implies that all states that we can reach from s in one cycle, including s_1, satisfy $\bigwedge_{i\in 0..k-2} \chi_i$, which is equivalent to ψ_{k-2}. This follows from (5.2). Moreover, from the fact that $s_1 \models \neg\chi_{k-1} \wedge \psi_{k-2}$ we know that the path from s to a state that violates output equivalence is exactly of length k. Each state on that path satisfies a different ψ predicate and, therefore, each state on that path is distinct.

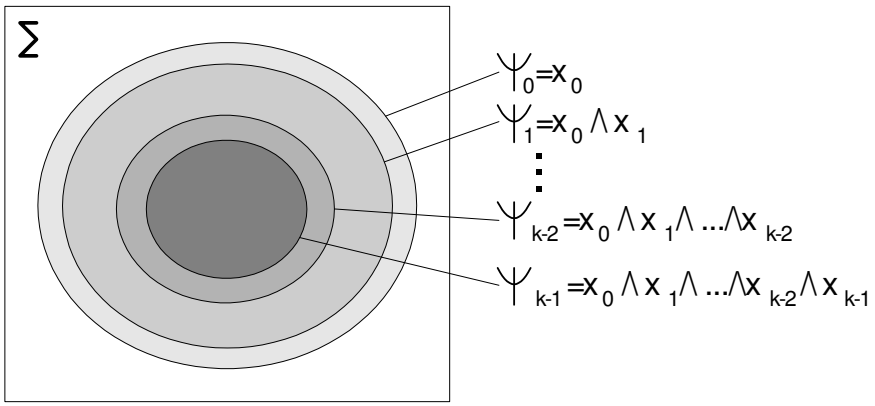


Figure 5.3. A Venn diagram of the states that can guarantee output equivalence for the next $0 \leq m \leq k-1$ cycles.

It is possible to enforce any circuit to satisfy the C-k-D predicate, e.g., by the method described in [4]. However, this is not the only method. Methods to achieve this goal can be applied not only on C, but also on the product $C_x = C \times C$ of the circuit with itself. In Section 5.5.1 we show a way to apply retiming and resynthesis without restrictions, which is independent of the method with which the C-k-D property was enforced.

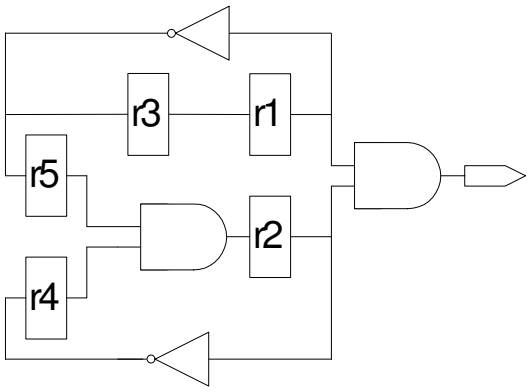


Figure 5.4. An example circuit.

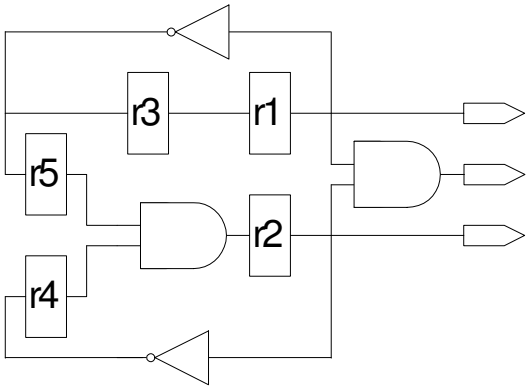


Figure 5.5. The circuit after the transformation.

As an example, consider the circuit of Figure 5.4. We take the product C_x of that circuit with itself and denote the registers of the first and second copy as $r1_a, ..., r5_a$ and $r1_b, ..., r5_b$ respectively. For C_x we have

$$\begin{aligned} \chi_0 &= (r1_a \wedge r2_a) \leftrightarrow (r1_b \wedge r2_b) \\ \chi_1 &= (r3_a \wedge r4_a \wedge r5_a) \leftrightarrow (r3_b \wedge r4_b \wedge r5_b) \\ \chi_2 &= (\overline{r1_a} \wedge \overline{r2_a}) \leftrightarrow (\overline{r1_b} \wedge \overline{r2_b}) \end{aligned} \tag{5.5}$$

States $s_a = [r1_a = 0, r2_a = 0, r3_a = 0, r4_a = 1, r5_a = 0]$ and $s_b = [r1_b = 1, r2_b = 0, r3_b = 0, r4_b = 1, r5_b = 0]$ satisfy $\psi_1 = \chi_0 \wedge \chi_1$, but not χ_2. In order to enforce ψ_1 as an invariant, we create pseudooutputs and connect to them the registers with a path of 2 registers to an existing output. The set of registers satisfying this requirement is $R_2 = \{r1, r2\}$. The new circuit can be seen in Figure 5.5. After simplifications, for the new product circuit

we have

$$\begin{aligned}\chi'_0 &= (r1_a \leftrightarrow r1_b) \wedge (r2_a \leftrightarrow r2_b)\\ \chi'_1 &= (r3_a \leftrightarrow r3_b) \wedge ((r4_a \wedge r5_a) \leftrightarrow (r4_b \wedge r5_b))\end{aligned}$$

States s_a and s_b do not satisfy χ'_0 and as a result they cannot satisfy $\psi'_1 = \chi'_0 \wedge \chi'_1$. It can be shown that ψ'_1 is an invariant of the modified product circuit and, therefore, the new circuit has the C-2-D property. Next, we formally describe the transformation.

Given a $k \in \mathbb{N}$ and a circuit C we modify the product $C_x = C \times C$ of the circuit with itself, so that formula (5.4) holds for C_x. This has the same effect as enforcing the C-k-D property on C. Assume that C_x is the product of the circuit with itself. We build ψ_{k-1} and check whether it is an invariant. This check can be formulated as

$$\forall (s_a, s_b) \in \Sigma_x \quad : \quad (s_a, s_b) \models \psi_{k-1} \;\Rightarrow\; (s_a, s_b) \models \chi_k$$

The reason we do not need to check $\bigwedge_{j \in 0..k-1} \chi_j$ is that we know that it is satisfied by formula (5.2) and the definition of ψ_{k-1}. We assume that k is chosen in such a way that the above check is tractable. If the formula above is false, then we know that the circuit C does not have the C-k-D property[1]. Then we chose $1 \leq m \leq k$, so that the number $|R_m|$ of registers connected by a path of m registers to an output is minimum. We create for each register $r \in R_m$ a pseudo-output with the name o_{r_m} and connect the register to the output.

[1]The case of a bug is not realistic, as we assume that C is sequentially equivalent to itself.

Let us denote as χ'_0, ψ'_{k-1} the predicates of the circuit after the first modification. For each state (s_a, s_b) with $(s_a, s_b) \not\models \chi_m$ we know that there must be at least one register in R_m that has a different value in s_a than in s_b. By connecting those registers to outputs, when we consider χ'_0, we have that $(s_a, s_b) \not\models \chi'_0$. Therefore, if we start again the process of deriving ψ'_{k-1}, this time ψ'_{k-1} implies ψ_{k+m-1}. If ψ'_{k-1} fails the invariant test, we repeat the same process after identifying another number m'. Then the next predicate ψ''_{k-1} tried implies $\psi_{k+m+m'-1}$. This process continues until ψ_{k-1} becomes an invariant. During this process the number of input variables for the computation of ψ_{k-1} remain bounded by $k \cdot |I|$ and the number of state variables are bounded by the number $|R|$ of registers in the product circuit. Since for the number m chosen in each iteration we have $m \geq 1$ and the diameter is an upper bound for k, eventually this process terminates. The advantage of this approach is that it is easy to apply as it does not require the BDD construction for ψ_{n-1} for some n. For example, if for some $n \in \mathbb{N}$ the BDD construction of ψ_{n-1} cannot be completed, by applying the transformation we can enforce ψ_{l-1} as an invariant with $l < n$. Then ψ_{l-1} could be easier to compute. The choice of k for the target invariant ψ_{k-1} should be made in such a way that ψ_{k-1} can be computed.

The modification we described is applied to the product circuit. However, it does not prevent us from obtaining the transformed circuit by applying a sequence of retiming and resynthesis operations. In Section 5.5.1 we show how we can apply these operations without restricting their optimization power or increasing their complexity.

5.4. Checking RnR Equivalence when the Original Circuit is C-k-D

In this section we show how we can check the equivalence between the original and the transformed circuit when we know that the original circuit fulfills the C-k-D property. The check can be done by unrolling the product circuit a bounded number of times. First, we prove some properties for the non-dangling states of the original C_a and the transformed circuit C_b. Using these properties we show that ψ_{k-1} is an invariant of those circuits in the non-dangling state space, if C_a satisfies the C-k-D property. Based on this result we prove the completeness of our method, i.e., if the checks fail, C_b cannot be obtained from C_a using retiming and resynthesis operations. Finally, we show how to check sequential equivalence between C_a and C_b.

Lemma 5.3. *If the original circuit C_a is RnR equivalent to the transformed circuit C_b, then for every non-dangling state of C_b there exists an equivalent non-dangling state in the STG of C_a.*

Proof. We prove the theorem by induction. The base case, before any step of the RnR sequence, is trivial as the transformed circuit is identical to the original circuit and the STGs are isomorphic. Assume that the lemma holds after m steps of the RnR sequence, we prove that it holds after the $m + 1$ step has been applied.

The first case is that the $m + 1$ step is a resynthesis step. Then the STG of the transformed circuit is preserved. Therefore, for every non-dangling state of C_b, there exists an equivalent non-dangling state in the STG of C_a.

The second case is that the $m+1$ is a retiming step. Let C_b^m, C_b^{m+1} be the transformed circuits before and after the m step, respectively. Retiming does not create new non-dangling states, but merges equivalent non-dangling states or splits non-dangling states to equivalent non-dangling states [39]. Consequently, for every non-dangling state in the STG of C_b^{m+1}, there exists an equivalent non-dangling state in the STG of C_b^m. □

Lemma 5.4. *If the original circuit C_a has the C-k-D property and the transformed circuit C_b is RnR equivalent to C_a, then ψ_{k-1} is an invariant of the product circuit $C_{a\times b} = C_a \times C_b$ in the non-dangling state space.*

Proof. Let us assume C_a has the C-k-D property and C_b is RnR equivalent, but ψ_{k-1} is not an invariant of $C_{a\times b}$. Then there exist k distinct states (s_1, t_1), ..., (s_k, t_k) in $C_{a\times b}$ such that

$$\begin{aligned}
\forall m \in 1..k-1 \quad &: \quad \exists i \in I : (\delta_a(s_m, i), \delta_b(t_m, i)) = (s_{m+1}, t_{m+1}) \\
\forall m \in 1..k-1 \quad &: \quad (s_m, t_m) \models \psi_{k-m} \\
(s_k, t_k) \quad &\models \quad \neg\chi_0
\end{aligned}$$

Since C_a has the C-k-D property, there is no state x such that $(s_1, x) \models \psi_{k-1}$ and $s_1 \not\approx x$. This implies that t_1 is either non-equivalent to any state of C_a or t_1 is equivalent to some state y of C_a for which $(s_1, y) \not\models \psi_{k-1}$. In the first case we have that t_1 is a dangling state or C_b is not RnR equivalent to C_a (Lemma 5.3), which is a contradiction. Therefore, t_1 is equivalent to y with $(s_1, y) \not\models \psi_{k-1}$. This implies that there exists $m < k-1$, such that $(s_1, y) \not\models \chi_m$. However, then there exists an input $i \in I^m$ for which $\lambda_b(t_1, i) = \lambda_a(y, i) \neq \lambda_a(s_1, i)$. Consequently, $(s_1, t_1) \not\models \psi_{k-1}$, which is a contradiction. □

Based on the lemmas above, if we are given an initial state (s_{ia}, s_{ib}) for the product circuit, we can use the following method

$$\exists s_a \in \Sigma_a, \exists i \in I^{n_d} \ : \ \delta_a(s_a, i) = s_{ia} \tag{5.6}$$

$$\exists s_b \in \Sigma_b, \exists i \in I^{n_d} \ : \ \delta_b(s_b, i) = s_{ib} \tag{5.7}$$

$$(s_{ia}, s_{ib}) \models \psi_{k-1} \tag{5.8}$$

$$\forall (s_a, s_b) \in \Sigma_{a\times b}, \forall i \in I^{n_d} \quad : \quad \delta(s_a, s_b, i) \models \psi_{k-1} \ \Rightarrow$$

$$\delta(s_a, s_b, i) \models \chi_k \tag{5.9}$$

where n_d is the register depth of the initial circuit. The first two formulas check that the initial states of the two circuits are non-dangling. They can be posed as SAT problems. The third formula checks the initial state of the product circuit satisfies ψ_{k-1}. Finally, the last formula checks that ψ_{k-1} is an invariant in the non-dangling state space. By considering only states reachable after n_d cycles, the check is restricted to non-dangling states.

If $C_{a\times b}$ satisfies formulas (5.6)–(5.9), then C_a and C_b are sequentially equivalent. This is because the product circuit starts from non-dangling states. Then dangling states are not reachable. Moreover, ψ_{k-1} is an invariant and it implies output equivalence by construction. The following theorem shows that if C_a is a C-k-D circuit and one of the formulas does not hold, then either C_a and C_b are not RnR equivalent or the initial states include dangling states.

Theorem 5.1. *If C_a satisfies the C-k-D property, the initial state of the product circuit is non-dangling and either formula (5.8) or (5.9) does not hold, then C_a and C_b are not RnR equivalent.*

Proof. Since the initial state of the product circuit is non-dangling, formulas (5.6) and (5.7) must hold. Then from Lemma 5.4, it follows that the circuits are RnR equivalent only if (5.9) holds. Moreover, if (5.8) does not hold, either dangling states are included in the initial state set (contradiction), or the circuits are not RnR equivalent. □

We believe that the assumption for (s_{ia}, s_{ib}) is reasonable. If s_{ia} is a dangling state, then the problem of finding a corresponding state after retiming may be unsolvable.

From Theorem 5.1 and the discussion above, we know that if (5.6)– (5.9) hold, then C_a and C_b are sequentially equivalent. If one of them does not hold, then C_b cannot have been obtained from C_a by a sequence of retiming and resynthesis transformations, i.e., C_b is not RnR equivalent to C_a. There may be a case that C_b is not RnR equivalent to C_a, but the two circuits are sequentially equivalent. In that case the checks for (5.8) and (5.9) may succeed or fail. In such a case a failure of the checks can point to an error of our RnR transformation implementation. A success is also a good result, because even though there may be a problem in the way the RnR transformation was implemented, the two circuits are sequentially equivalent. Ideally, in this case we would like to get both results. Our method gives only one of the results.

5.5. Applying Retiming and Resynthesis without Restrictions

In Section 5.3 we presented methods to check whether a circuit C has the C-k-D property and transform the product circuit $C_x = C \times C$, so that property ψ_{k-1} becomes

an invariant. In this section we show how we can apply retiming and resynthesis on C without restrictions from that transformation (Section 5.5.1). More specifically, with our method neither the optimization power of retiming and resynthesis is reduced nor their complexity is increased. Our method is independent of the way the modified circuit is obtained. It guarantees that after the retiming and resynthesis for some $m \in \mathbb{N}$ the predicate ψ_{m-1} is an invariant of the product circuit after the synthesis. Finally, we show how to obtain the transformed circuit after verification (Section 5.5.2).

5.5.1. Method to Apply Unrestricted Retiming and Resynthesis

We assume that we are given the modified product circuit C_x that has been augmented with additional gates and outputs by a method enforcing the C-k-D property, e.g., the method of Section 5.3.3. We also assume that in C_x we can distinguish the two copies of C, namely, C_1 and C_2, and the additional logic and outputs C_3. More specifically, each node and edge of the product circuit is colored from the set $\{c_1, c_2, c_3\}$ based on its origin. A c_3 edge can be driven by a gate of any color. However, a c_1 or c_2 edge can be driven only by a node of the same color. Predicate ψ_{k-1} is an invariant of C_x. Our purpose is to apply a sequence of retiming and resynthesis transformations on C_2, the second copy of C, without being restricted by the additional logic. Moreover, after the transformation we want the product circuit to have ψ_{m-1} as an invariant for a known m.

Before a resynthesis step we extract C_2 from C_x by considering all circuit nodes and edges marked with c_2. All c_3 edges that are driven by a c_2 node are left hanging, i.e., not driven by any node, by this transformation. We express each of these edges as a function of c_2 registers and primary inputs. Then we add logic to C_3, so that the only edges

Bench	SAT on miter with ABC			
	Without Addit. Logic		With Addit. Logic	
	Exec Time (sec)	k	Exec Time (sec)	k
$s635$	4	>4	4	2
$s838$	10.6	>4	6	3
$s938$	7.4	>4	5.4	2
$s953$	437	>4	3.6	2
$s967$	651	>4	5.6	2
$s1196$	3.7	2	3.4	1
$s1512$	>2000	>4	27.25	2
$s3271$	>2000	>4	>2000	2

Table 5.1. Running-time and k value for which ψ_k becomes an invariant. For the SAT on the miter case, the computation is terminated if the running-time is greater than 2000 secs, or k is greater than 4. Enforcing ψ_{k-1} as an invariant can significantly speed up sequential equivalence checking in some cases.

Bench	VIS: backward traversal		VIS: forward traversal	
	Without Addit. Logic (sec)	**With Addit. Logic** (sec)	**Without Addit. Logic** (sec)	**With Addit. Logic** (sec)
$s635$	> 2000	1.2	> 2000	> 2000
$s838$	74.6	15.6	1.7	4.2
$s938$	>2000	0.75	> 2000	> 2000
$s953$	4.41	1.28	1.4	1.4
$s967$	3.7	3.7	2.9	2.5
$s1196$	1.1	1	0.9	0.8
$s1512$	> 2000	21.28	> 2000	> 2000
$s3271$	> 2000	> 2000	> 2000	> 2000

Table 5.2. Experimental results with vis (backward and forward traversal) with and without the additional logic that enforces ψ_{k-1}. The computation is terminated if the running-time is greater than 2000 secs. Enforcing ψ_{k-1} as an invariant can significantly speed up sequential equivalence checking in some cases.

hanging, i.e., not driven by a node, are the edges that would be driven by a C_2 register or a primary input. It is easy to extract the additional logic by traversing backwards a c_2 node that drives a c_3 edge until a register or a primary input is met in each path. Then

this logic is replicated and added to C_3. Then we apply resynthesis on C_2. The resynthesis optimization is unrestricted as only nodes and edges of C_2 are considered. Resynthesis does not remove registers or inputs and after the step we can bring the modified version of C_2 back in C_x by connecting c_2 nodes (registers, inputs) to the corresponding hanging c_3 edges.

Before a retiming step we extract C_2 from C_x again. We maintain a mapping between the hanging c_3 edges and the c_2 nodes that drive those edges. We retime C_2 as an independent circuit. Retiming does not change the circuit structure, so we reconnect the modified C_2 to obtain C_x by preserving the mapping. Nodes belonging to C_2 that drive c_3 edges may have a lag value r that is different than 0. In such a case, the number of registers on the c_3 edges need to be adjusted, so that the weights of the edges are consistent with the lag (r-) values.

If for a c_2 node v that drives a c_3 edge we have $r(v) < 0$, then we add to the c_3 edge $-r(v)$ number of registers. These registers have been moved across v from its fan-in, which are c_2 edges. Therefore, this move is based on pre-existing registers. Based on the results proved on Section 5.4, we know that after such a retiming move ψ_{k-1} should still be an invariant of the product circuit C_x in its non-dangling state space.

In the case that $r(v) > 0$ for a c_2 node v driving a c_3 edge, then we remove $r(v)$ registers from the c_3 edge. If the weight of the edge becomes negative, we try to adjust the $r(u)$ value of the head u of the edge, which is a c_3 node. This may cause other edges to have negative values. The paths from all these edges terminate at a single pseudo output o[2]. The end result may be that we have to adjust the value of the pseudo output

[2]We call o a pseudo output because it is added by our method and it is not an output of the original circuit.

o. In that case the effect of the additional logic is moved in the past. Therefore, instead of ψ_{k-1} the predicate that must be an invariant of the product circuit is $\psi_{k-1+r(o)}$. Every time we retime a pseudo output, we adjust the number m, for which ψ_{m-1} must be an invariant of the modified product circuit. Using this number, at the end of the retiming and resynthesis sequence, we will derive ψ_{m-1} and require that the checks described in Section 5.4 succeed, in order for the two circuits to be sequentially equivalent.

The method described in this section resembles recording the transformation history of retiming and resynthesis. In [66] a similar method is presented. There are important differences between that approach and our method. First, in [66] verification relies on synthesis to record the candidate problems to be solved. If the problems are solved successfully, then the circuits are assumed equivalent. However, it is unclear whether a bug in recording synthesis history can result in a false positive. Moreover, that technique is described for a tool that uses a specific data structure to represent a logic network, namely And-Inverter-Graphs. Our technique is general in that the data passed from synthesis to verification are in the form of a circuit. Therefore, synthesis and verification can use different data structures. In terms of efficiency, our approach creates new nodes only when required for the fan-in cones of the pseudo-outputs, while the approach in [66] stores every node created during synthesis. Moreover, it stores one node for each move of a register over a gate during a retiming operation. Because of that, restrictions on the synthesis part may become necessary for the approach in [66] to be practical.

5.5.2. Method to Obtain Transformed Circuit

The product circuit obtained by the method of the previous section can be used for verification purposes. However, after verification we want to extract only the optimized copy C_2 of the product circuit. It is easy to see that by taking only the nodes and edges colored by c_2, we have the optimized version. The extracted C_2 circuit is the same as the circuit obtained by applying retiming and resynthesis to C_1. The reason is that during the optimization the additional edges and nodes were not considered.

5.6. Experimental Results

In this section we present experimental results for our approach. We used the ABC framework [11] and VIS [14] to test our ideas. Checking formula (5.9) is the most difficult step of our approach, so we focus on the implementation and results for that part. With ABC we implemented the check as a SAT problem on a miter. More specifically, the predicate ψ_{k-1} is built as a BDD after unrolling the product circuit and applying universal quantification on the inputs. The result is appended to the circuit specifying χ_k using the utility of ABC that implements a BDD as a circuit of muxes. Then we check whether for any input sequence and state $\psi_{k-1} \wedge \neg\chi_k$ is satisfiable [3].

We tried to verify a number of ISCAS benchmarks with and without additional logic that enforces the C-k-D property (Tables 5.1 and 5.2). The pairs of circuits that we tried are retiming and resynthesis equivalent. The first two columns display the running time and the k value of the verification procedure without the additional outputs and logic.

[3]The approach in [10] for building the BDD for target enlargement or the SAT approach for quantification [65] could be more efficient for checking formula (5.9). However, we do not expect that another method will significantly change the improvement shown in the results and the conclusions drawn by them.

The value k is the number for which ψ_k becomes an invariant. The next two columns show the same results when the additional outputs were included in the circuits. Except for the additional outputs and the logic driving them the pairs of circuits are in both cases the same. The circuits with and without the additional logic were then checked with VIS. We tried both backward (columns 5 and 6) and forward traversal (columns 7 and 8) traversal with VIS. The VIS-command we used on the circuits is "seq_verify" with option "-r" for variable reordering.

In many cases the running time of the procedure is substantially reduced. Examples are the cases of s635, s938, and s1512. VIS does not terminate in 2000 secs without the additional logic. However, the computation of ψ_{k-1} with additional logic takes a few seconds in those cases. Enforcing ψ_{k-1} can speed up the verification in VIS, as well, when it operates in backward traversal. The additional logic does not have any significant effect for the forward traversal. For the check of ψ_{k-1} as an invariant, the speed up obtained by enforcing a small k is significant in most cases.

From the results we conclude that by using the additional logic we can significantly speedup the checking procedure in some cases. The described procedure is based on BDDs and, therefore, does not scale well compared to SAT based procedures, e.g., [40]. However, those procedures are not proved complete for retiming and resynthesis sequences with more than one resynthesis step. We conclude that enforcing the C-k-D property can simplify the verification task without restricting the synthesis part.

5.7. Summary

In this chapter we extended the property of C-1-D to C-k-D and we showed how we can check circuits for equivalence if one of them satisfies the C-k-D property. We also presented a technique to enforce the C-k-D property on a circuit and then apply a sequence of retiming and resynthesis transformations without restricting their optimization power or increasing their complexity. Our method provides a bound to the number of timeframes that need to processed during verification and is complete in the sense that any result provides useful information. Our experimental results show that enforcing the C-k-D property can speed up the verification process. In the next chapters we will show how abstraction can be used to simplify the verification problem.

CHAPTER 6

An Efficient System-Level to RTL Verification Framework for Computation-Intensive Applications

6.1. Introduction

It is known that almost two thirds of the design cycle of a digital integrated circuit is spent on verifying its functionality [49]. As designs become more complicated and time-to-market periods decrease, the needs for efficient verification frameworks increase.

For verifying digital integrated circuits several approaches exist. One is the simulation based verification, whose drawbacks are the long execution time and the inability to assure correctness of the design. On the other hand, formal verification can be an efficient alternative to prove that specific properties of the design hold.

Formal verification methods include Symbolic Model Checking and Theorem Proving. In Model checking [22], the temporal logic specification is used to check system properties where the system is modeled as a finite state machine. Symbolic Model Checking, with boolean encoding of the finite state machine as ordered binary decision diagrams (BDDs) can handle more than 10^{20} states [64]. On the other hand, Bounded Model Checking (BMC) for linear temporal logic (LTL) can be reduced to propositional satisfiability in polynomial time where bound is the maximal length of a counterexample and solved using SAT solvers [13].

For Theorem provers both the system and its desired properties are expressed as formulas in some mathematical logic and the theorem prover finds a proof from axioms of the system. SVC [8] and its enhanced version CVC [81] are automatic theorem provers for first order logic. PVS [72] combines decision procedures and model checking with interactive proof. Theorem provers in contrast to model checkers can handle infinite state spaces but generally require manual intervention and are hard to use.

It is a common practice to write a System Level functional description of the digital IC in the first design stages. This description after verification can be used as the golden model for the Register Transfer Level implementation of the circuit. One approach for this kind of verification is CBMC [19], which uses a C specification of the circuit to verify the RTL model. The techniques to capture the model are the same as in BMC approaches and a bit-level SAT solver [67] is used to produce a counterexample or to prove the correctness of the assertions.

In this chapter we describe an alternative approach to CBMC for verifying properties of an RTL description using its System Level specification. The approach is orders of magnitude faster than CBMC for computational intensive applications by sacrificing bit-level accuracy, which may not be needed during the early stages of the verification process. Morevoer, if the implementation of arithmetic operations is the same in the golden model and the RTL description, a verification procedure working at the bit-level may not be needed. The back-end tool used in our framework is Mathematica, a well known commercial symbolic analysis tool. In the next section we describe the motivation behind this approach. In Section 6.3 we formulate the problem that needs to be solve, while in Sections 6.4 and 6.5 we explain the reasons behind using Mathematica and the

proposed framework in more detail. Finally, Sections 6.6 and 6.7 present our results and conclusions.

6.2. Motivation

In this section we discuss the motivation behind the usage of word level techniques and Mathematica. As mentioned before, CBMC is the only bounded model checking approach for verifying properties of an RTL implementation based on a System Level description. CBMC uses a SAT solver as the back-end tool for proving the assertion or providing a counterexample. The whole program needs to be converted in a Conjunctive Normal Form (CNF), which will be the input to the SAT solver [67].

A major bottleneck for the SAT solver is the memory requirements of a CNF. If the required memory exceeds the available physical memory, the swap file will be employed. Therefore, all non-local accesses for the formula will involve the disk and become very expensive. There are two factors that will determine the size of a CNF, the number of clauses and the number of literals.

The number of clauses depends on the logic that the circuit will implement. Arithmetic operations like addition, or multiplication produce a large number of clauses [20]. A behavioral System Level description of a computational intensive application is expected to include mainly arithmetic operations. Therefore, for computational intensive applications the number of generated clauses can make the usage of a SAT solver inefficient.

Besides the number of clauses, the number of literals of a CNF will impact the running time and depends again on the logic of the circuit. The arithmetic operations affect that number depending on the type of operators and the size of input operands. Moreover,

the number of literals increases with the number of cycles for which the model checking is performed. However, given the area constraints in the synthesis of digital circuits and that significant properties require a large amount of cycles to be proved, the bound in the number of cycles cannot be small for realistic designs and that can lead to an explosion in the memory requirements. As an example in [19] for a DRAM Arbiter, which is not expected to have many arithmetic operations, the memory usage was 2GB for sufficiently large bounds.

The above discussion reveals the need for a higher level of a abstraction for the System Level and the RTL models used for verification. Especially, for computational intensive applications verification techniques at the word level would be extremely useful for the reasons mentioned above. By representing the two models using a conjunction of word level formulas many important properties of a computational intensive description can be verified, as will be shown in the next sections. The cycle accuracy of the RTL description can be maintained, however the bit accuracy will be lost. The bit accurate implementation of the design can verified later in the design cycle when word level design faults have been eliminated.

Approaches that have already used word level techniques for verification problems include [72],[8], [81]. None of these approaches can handle multiplication of two variables in an efficient way. In [24] the authors used SVC [8] to verify assembly routines used in DSP software. Because SVC uses uninterpreted functions to handle all operations except addition and subtraction, the authors had to support user-defined properties, which would specify for example the multiplication is a commutative operator. Moreover, heuristics were written for operand reordering and expression normalization.

Since computational intensive applications are expected to have a large amount of complicated arithmetic expressions, the designer would need to define all the important properties for multiplication and division. Then the properties that combine two or more operators also have to be specified, like $a(b+c) = ab+ac$. Furthermore, the heuristics for simplifying or reducing complex arithmetic expressions should be as complete as possible for these applications. Obviously, the manual effort for verifying computational intensive applications using these word level tools is large and makes their usage for this kind of verification problems inappropriate.

In conclusion, there is a need for another approach to functional verification of computation intensive applications. This approach should be able to prove functional correctness by bringing variables to the infinite domain, so that the verification procedure will be fast and should use word-level provers that will not require manual effort from the user for defining the properties of commonly used arithmetic operators like non-linear multiplication.

6.3. Problem Formulation

In this section we formulate the problem that we try to solve. Given a System Level and a synchronous, single-clocked RTL description, with a mapping of corresponding input variables, a bound in terms of clock cycles, and an assertion inserted in the code, we try either to prove that the assertion is true for that specific bound and for all valid input values, or find a counterexample that can make the assertion false. We try to solve the problem by bringing all variables to the infinite-precision domain.

6.4. Mathematica

Mathematica [87] is a technical computing package with a very rich library for symbolic computation. It includes solvers for sets of formulas in different domains (Boolean, Integer, Real) and has built-in the properties of the most commonly used arithmetic operators of each domain. The advantage of Mathematica is that is a generic package and, therefore, the user will not have to spend manual effort to express the properties of word level operations. Moreover, it is a powerful symbolic analysis tools and supports solvers at the integer and real domain, avoiding the translation of arithmetic expressions to binary numbers. Finally, arithmetic expression manipulation with Mathematica has been used efficiently in the past for several compiler problems [41], [42].

6.5. Verification Framework

In this section we will present the EVRM (Efficient VeRification based on Mathematica) framework. A schematic representation of EVRM is shown in Figure 6.1. As it can be seen from this figure the framework can be broken into three parts. The first and second parts implement the transformation of the System-Level and RTL descriptions to word level expressions. The third part generates the Mathematica statements from these expressions. Then these statements are going to be the input to the Mathematica Kernel, which is going to either prove the validity of the asserted property or provide a counterexample, for which the property is invalid. In the next sections we describe each part of the framework in more detail.

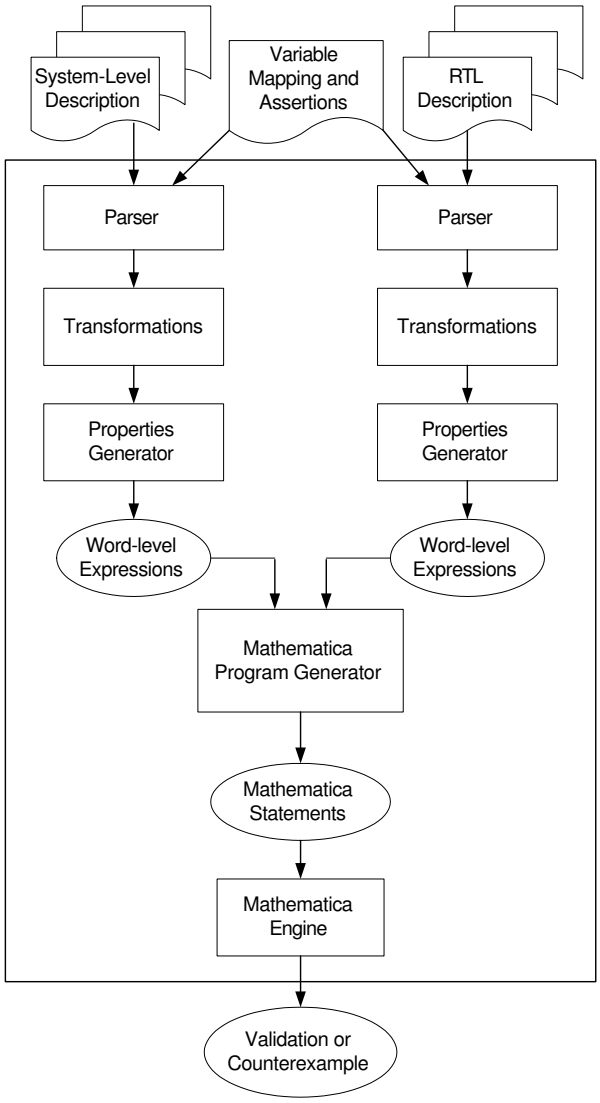


Figure 6.1. The EVRM framework.

6.5.1. System-Level description to Word-Level Expressions

The SL description is transformed to word level expressions using the techniques presented in [20]. Currently, the input SL description should be in ANSI C. The first step is to unwind all loops in the code and to transform switch statements to if statements. Then

the code is converted in a Static Single Assignment (SSA) form. At the end of the transformation phase the code consists of assignment statements and if statements.

6.5.2. Register Transfer Level description to Word-Level Expressions

For this process we assume that the RTL description represents a Finite State Machine of a single clock design. The process starts by finding the registers of the circuit. Then the program is transformed to a set of assignment statements and if statements by modifying the control structures of the description. This process is described in detail in [20].

6.5.3. Word-Level Equations

After transforming the two descriptions in sequential programs that contain only if statements and assignments, the generation of guarded word-level equations starts.

A guarded word-level equation has the form:

$$(cond)\ ?\ var = Expr1\ :\ var = Expr2$$

where *cond* is a binary variable that represents the conjunction of all conditions that should be true, in order for the assignment to be executed. These are the conditions of the nested if-statements, in the blocks of which the assignment is placed. The conditions are represented by:

$$ExprLeft\ \ Operator\ \ ExprRight$$

where *Operator* can be any of the $>, <, ==, !=, >=, <=$ and $ExprLeft$, $ExprRight$ are arithmetic expressions consisting of constants, variables, parenthesized subexpressions, and arithmetic operators, like $+, -, *, /$. The same applies for $Expr1$ and $Expr2$ of the

guarded word level equation. In the guarded word level equation *var* is the name of the variable that is assigned.

All guarded word-level expressions are later transformed to Mathematica statements. The above generic guarded word level expression will be translated to:

$$\{cond \,\&\&\, var == Expr1\} \,||\, \{!(cond) \,\&\&\, var == Expr2\}$$

The variables of the above expressions can have several types. Most commonly used for DSP applications are Integers and Reals. Variables may also have compound types and indexed multi-dimensional arrays can be used in all arithmetic expressions.

6.5.3.1. FindInstance statement. After all expressions, which represent the constraints imposed by the two descriptions, are transformed to equivalent Mathematica expressions, the asserted expression will also be transformed. Then the conjunction of the all constraints and the negation of the assertion will be the expression that if it is satisfiable then the asserted property does not hold. Satisfiability of this expression means that there are inputs that satisfy all constraints imposed by the descriptions of the specification and the implementation and satisfy the negated property as well. Therefore, the property is not valid for all inputs.

Mathematica version 5.0 has a statement that allows the user to find an instance of a set of variables that makes an expression valid. The syntax of the generated statement will be:

$$FindInstance\,[\,\underbrace{E_1 \,\&\&\, E_2 \,\&\&\, ... \,\&\&\, !A}_{expression}\,,\,\overbrace{\{a, b, c, ...\}}^{variables}\,,\,D\,]$$

The first argument is the expression which may become true for an instance of the set of variables given as a second argument. The third argument D is the domain to which the variables belong. The domain can be Integers, Reals, or Booleans in this case. As explained previously the generated statement by EVRM sets as first argument the conjunction of the constraints $(E_1, E_2, ...)$ and the negated assertion A. The set of variables given as the second argument are the set of all inputs and variables of the description.

The $FindInstance$ exits either by reporting the empty set, in which case it has proven that there are no values for the variables that can make the expression hold, or by reporting values for the variables that will make the expression hold.

6.5.3.2. Optimizations. In the CBMC approach the SL and RTL descriptions were translated to a CNF. Bit vector arithmetic operations in this case were transformed to CNF by using actual circuit gate representation. Since these circuits normally work on two input variables, there was no way to reduce the problem instance by replacing variables with their definition. However, in our case replacing a variable with its actual definition can reduce the size of the input expression and the cardinality of the variables set for the $FindInstance$ statement.

So, if $a == b + c$ is an input word level expression with $guard == TRUE$, then it was transformed to a statement $a = b + c$, which was executed before $FindInstance$. Moreover, $a == b + c$ could be removed from the set of constraint expressions and did not need to be part of the first argument of $FindInstance$, since it was forced. Added to that, a was removed from the set of variables, as Mathematica would replace every

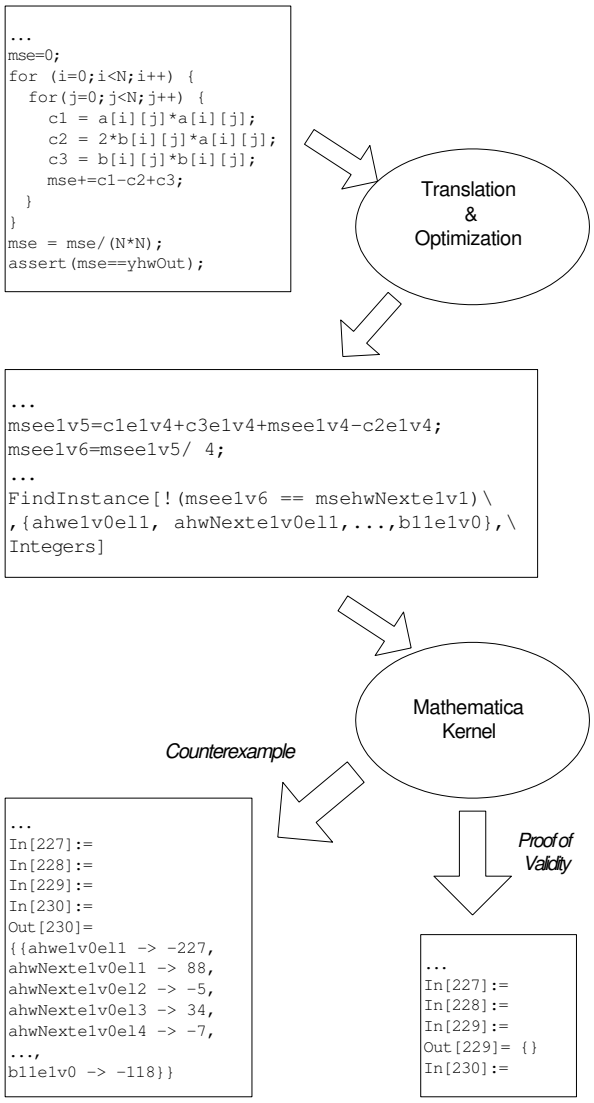


Figure 6.2. Example demonstrating the code generated by the framework.

instance of a with $b + c$. This optimization was applied to non-guarded expressions and reduced the problem instances.

Figure 6.2 shows some example instances of the generated statements and the output of Mathematica for a part of a C program.

Application	Matrix Mult	FIR	Laplace	Sobel	MSE
System-level lines	27	15	19	24	19
RTL lines	185	106	129	146	97
Chaff input clauses	1,034,271	145,635	8,964	28,885	339,010
Chaff input literals	308,158	44,024	4,547	10,332	101,206
Mathematica input expressions	3,701	1,037	294	303	222
Mathematica input variables	99	51	165	170	33

Table 6.1. Characteristics of the applications that were used for the experimental results.

Application	Matrix Multiplication	FIR	Laplace	Sobel	MSE
CBMC	> 5 h	> 5 h	> 5 h	> 5 h	> 5 h
EVRM	5:06 min	1:14 min	10:27 min	44:40 min	0:41 min

Table 6.2. Time to Prove Validity of the Assertion.

Application	Matrix Multiplication	FIR	Laplace	Sobel	MSE
CBMC	14:52 min	> 5 h	> 5 h	> 5 h	40:27 min
EVRM	5:23 min	1:17 min	0:37 min	7:07 min	0:40 min

Table 6.3. Time to Produce Counterexample.

6.6. Experimental Results

In this section the experimental results for 5 benchmarks will be presented. The characteristics of each benchmark are summarized in Table 6.1. The system-level specification of each application was written in C. The RTL implementation was described in RTL C. All results were taken on SUN Ultrasparc machines, with OS Solaris 9, and 256MB of physical memory. As explained above, the results are compared to CBMC as existing word-level solvers,to the best of our knowledge, cannot handle frequently used arithmetic operators without manual effort for the specification of the operators' properties.

Matrix Multiplication. This application implemented the multiplication of two 3x3 matrices. The assertion was checking whether the element of the last row and last column of the resulting matrix was the same for the two descriptions. For the system-level description the input values were stored in 2 dimensional matrices, whereas in the RTL implementation they were stored in single dimensional register arrays. Moreover, the elements of the resulting matrix were computed in a column-wise manner in system-level and in a row-wise in the RTL description. Added to that, the structure of the code was completely different for the two descriptions. In the system-level specification a 3-deep loop nest was implementing the multiplication after the initialization stage, compared to an 18-state RTL implementation, which produced the multiplication result after 140 cycles.

As shown in Table 6.2 for a correct version of the RTL implementation, it took almost 5 minutes for the EVRM framework to prove correctness for any two input matrices. This time is broken into two parts, the time to generate the statements and expressions from the two descriptions and the time it takes for Mathematica to read these statements and expressions and produce a result. The generation part took 4 minutes and the Mathematica part took 1 minute. With CBMC after 5 hours no result could be produced.

In the second experiment done with this application a bug was inserted in the RTL implementation. The bug was in the initialization of one of the variables. As shown in Table 6.3 it takes almost 5 minutes for EVRM to produce a counterexample, while for CBMC the time is close to 15 minutes. In case the matrices were larger, like 4x4 or 5x5 the difference could be larger as the memory requirements of CBMC would increase.

FIR filter. The next application used was a 4-tap FIR filter. The assertion was checking for the equivalence of the last element of the output array. The RTL description was implemented as an FSM of 12 states and produced the result after 65 cycles.

EVRM proved the equivalence of the two descriptions in less than 2 minutes, out of which 1 minute was spent in the generation of the statements and expressions for Mathematica.

In the second experiment for this application a bug was added for the output result. This time the bug would affect only a subset of the possible output values. The bug forced the output values for the RTL implementation not to exceed the value 8192, whereas for the system-level specification the result could be any integer value. EVRM found a counterexample after 1 minute and 17 seconds. In both cases the running time with CBMC was more than 5 hours.

Laplace Transform. This application implemented one step of the Laplace transformation. The values were stored in an two dimensional array in the System Level description, whereas in the RTL they were in a one dimensional array of registers. Moreover, the order of the computations was different in the two models.

EVRM proved that the output value of the two descriptions was equal for all inputs in almost 10 minutes. For the faulty version of the same algorithm a counterexample was found in less than a minute. The fault was the removal of the condition that did not allow the array elements to exceed the value 255 in the RTL description. For CBMC the bound of 5 hours was reached before an answer was given by the program.

Sobel Transform. This application was implementing the Sobel Transform. The assertion checked to whether the computation of the new value based on its eight neighbor

points was correct for the RTL implementation. It took almost 45 min for EVRM to prove the validity of the property.

The second experiment was done with a faulty implementation. The original RTL implementation was implementing the abs function by checking if the difference for the horizontal sobel transform was negative and in that case multiplying by -1. This was deleted in the faulty implementation allowing negative results of the difference. Again this bug will not be visible for all inputs as many will produce positive values for the horizontal difference. It took about 7 minutes for EVRM to find a counterexample that invalidated the assertion.

In both cases after 5 hours CBMC did not produce any result.

Mean Squared Error Computation. The last application was implementing mean squared error computation. The input to the algorithm were two 2x2 matrices for which the mse value was computed by the SL and RTL descriptions. EVRM could prove correctness for this property in time less than a minute.

The second experiment with this application involved the addition of bug in the RTL description. The inserted bug affected the initialization of a variable. Again EVRM found a counterexample after 40 secs. In this case CBMC provides a counterexample for the assertion after 40 minutes.

6.7. Summary

In this chapter we have presented a new approach for the verification of specific properties of an RTL implementation based on an executable System Level specification. The new framework is intended for computational intensive applications and is based on word

level techniques and uses Mathematica for the satisfiability procedure. The current approaches for the same problem are based on bit level SAT solvers.The results show orders of magnitude of performance improvement compared to CBMC, a tool used for RTL to SL verification. The next chapter provides abstraction techniques that can be used for the verification of self-stabilizing systems, i.e., systems that recover from any transient fault.

CHAPTER 7

Abstraction Techniques for Parameterized Self-Stabilizing Systems

7.1. Introduction

Automated methods for the verification of distributed systems can only be applied to relatively small finite-state systems. However, most distributed algorithms are specified for an arbitrary number of processes. More specifically, the number N of processes present in the distributed system is a parameter and the algorithm is expected to work for any valid value of the parameter. We call these systems parameterized systems. An instance of the parameterized system is the system built for a specific value of N. Although automated methods, i.e., model checking, can be used for the verification of instances with small number of processes, they can neither be efficiently applied to large instances nor prove that all possible instances of a system are correct. In those cases abstraction is necessary.

Using abstraction a finite-state system can be derived from a parameterized system. We call the derived system the abstract system. If the correctness condition holds for the abstract system, then it holds for all instances of the parameterized system. Since the state space of the abstract system is finite, model-checking can be used to check whether it satisfies the correctness condition. Deriving an abstract system requires deep knowledge

of the specification of the system and model checking and, therefore, general abstraction methods are desirable.

A number of abstraction methods have been developed for high-atomicity parameterized systems [21, 74, 9, 30]. High-atomicity parameterized systems are systems in which the number of variables each process can read or write in one atomic step increases, as the parameter N increases. Since for large N such communication operations become very expensive, we focus on low-atomicity systems.

Our work targets a specific class of fault-tolerant systems; self-stabilizing systems. Self-stabilizing systems are systems that automatically recover after any transient fault [27]. For those systems liveness properties, i.e., properties that specify that something good will eventually happen, are more relevant than safety properties, i.e., properties that specify that nothing bad will happen. This is because transient faults can bring the system in any arbitrary state, making all states in the state space reachable (Section 7.2). Since we focus on self-stabilizing systems, we consider only abstraction methods for the verification of liveness properties.

For the verification of liveness properties in low-atomicity parameterized distributed systems two abstraction techniques have been developed: the method of invisible ranking [33] and the method of control abstraction [52, 43]. The idea behind the method of invisible ranking is to bound the number of processes needed to prove a correctness property for a class of parameterized systems [33]. The approach can be used for the verification of properties of the form $\Box(p \rightarrow \Diamond r)$, i.e., for every state satisfying assertion

p there is a future state satisfying assertion r. It is not known how other liveness properties can be checked using this method. Moreover, in some cases the number of required processes is large (128 for the dining philosophers problem).

An alternative approach is the method of control abstraction. The idea behind control abstraction is to abstract away an arbitrary number of symmetric processes by using a particular process called network invariant. Then the correctness property is checked in the abstract system, which is composed of a small finite number of processes and the network invariant [43]. There are two difficulties that have restricted the applicability of this method. The first is that there is no automated method for the construction of the network invariant. Existing automated approaches for the construction of network invariants target only safety properties [58]. The second is that the network invariant must have the same set of observable variables as the system of symmetric processes abstracted by it. As an example, consider a system with N processes in which each process has one variable that can be read by all other processes. In such a system the number of variables that each process can read increases with N and, therefore, a network invariant with a fixed set of variables cannot be constructed. Because of this constraint, the usage of control abstraction has been restricted to ring topologies of processes [45], in which each process reads the variables of only two neighbors. It has also been successfully applied on systems where the number of shared variables does not increase with the number of processes. An example is a mutual exclusion algorithm with all processes sharing only one semaphore [43].

In this chapter we present an abstraction technique that builds on the theory of control abstraction. The proposed technique is split into 3 parts. Each part is independent

and can be used as a stand-alone transformation. To the best of our knowledge, this is the first abstraction technique that can be used to prove the correctness of low-atomicity, parameterized self-stabilizing systems, whose number of observable variables may increase with the number of processes in the system. The case studies demonstrate that our abstraction technique is not trivial and can be applied to distributed algorithms to which no other abstraction technique has been successfully applied. Moreover, sufficient conditions under which the abstraction technique is complete are established.

The derived abstract system is relatively small and its state space does not increase exponentially with the number of states of the abstracted symmetric processes, as it is the case in [74]. The proposed abstraction technique handles both weak and strong fairness constraints for the abstracted processes, as opposed to previous works [9]. Finally, because it uses syntax manipulation, the complexity of the algorithms building the transition relation is low compared to approaches that use decision procedures (MONA) [74, 9].

We give a short description of self-stabilizing systems in Section 7.2. In Section 7.3 we describe the notation we use and the systems we consider. Section 7.4 gives an overview of the proposed 3-step abstraction technique and Sections 7.5,7.6, and 7.7 explain each step in detail. We demonstrate the application of the technique to a number of self-stabilizing systems in Section 7.8.

7.2. Self-Stabilizing Systems

Self-stabilizing systems are systems that can automatically recover from any transient fault [27]. This type of fault-tolerance is desirable in many distributed systems [6, 51, 86].

We distinguish two types of self-stabilizing systems; "strict-stabilizing" and "pseudo-stabilizing" systems [16]. After a fault, a strict-stabilizing system will eventually enter a state in which it satisfies the correctness property and starting from that state it cannot violate the correctness property anymore. Formally, let ϕ denote the correctness property, a strict-stabilizing system satisfies the LTL (Linear Temporal Logic) property $\Diamond\phi$ and $\Box(\phi \rightarrow \Box\phi)$, starting from any fault state. A pseudo-stabilizing system will eventually get into states where the correctness property will never be violated. The LTL property is given as $\Diamond\Box\phi$. Although pseudo-stabilization is weaker, it is sufficient for many practical applications.

System designers reason about the correctness of self-stabilizing systems by considering all states in the state space as initial states. The assumption is that the initial state is the first state after a transient fault. Therefore, the system can start from any state and must eventually recover, if no more faults occur. A common proof method for self-stabilizing systems is the method of convergence stairs [27, 34]. A finite sequence of predicates is defined $p_0, ..., p_m$ with $p_0 =$ True and $p_m = \phi$ being the correctness property of the system. Then the designer proves that in the system if eventually always p_i is satisfied, then eventually always p_{i+1} is satisfied for all $i \in 0..m-1$. In LTL, it is $\Diamond\Box p_i \rightarrow \Diamond\Box p_{i+1}$. By this method, the pseudo-stabilization $\Diamond\Box p_m$ (or persistence property) can be proved. In addition, by showing that the safety property $\Box(p_m \rightarrow \Box p_m)$ holds, strict-stabilization is also established. Proving the liveness properties $\Diamond\Box p_i \rightarrow \Diamond\Box p_{i+1}$ is the hardest step of this method and, therefore, we focus on this type of properties in this work. Unfortunately, most self-stabilizing systems are complicated and proving their correctness manually is not

an easy task. Therefore, there is a need for enabling the usage of automatic verification techniques for these systems.

7.3. Systems and Notations

A parameterized system is composed of N identical processes. Each process $P(i)$ is an instantiation of a generic specification $P(id)$ with $id = i \in 1..N$. The ids of the processes are used for naming convenience and no relation is specified over them. The generic process is defined as $P(i) = (V_i, W_i, V_{L(i)}, \Theta_i, \rho_i, L_i)$, where

- V_i is the set of all variables that process i can read or write.
- W_i is the set of owned variables that process other than i can modify ($W_i \subseteq V_i$).
- $V_{L(i)}$ is the set of local variables that no process other than i can read or modify ($V_{L(i)} \subseteq W_i$).
- Θ_i is a predicate over the variables in V_i that specifies the set of initial values that the variables can have.
- ρ_i is the next state relation.
- L_i is the liveness property.

We assume that the domains of all variables are finite. Moreover, the size of W_i is finite and independent of N.

A set S of processes is called composable if there is no process in S that can read a local variable of another process or write a variable owned by another process [43]. From the composable set S of processes we can create new processes by the parallel asynchronous composition of some processes in S. Let $A \triangleq (V_A, W_A, V_{LA}, \Theta_A, \rho_A, L_A)$ and $B \triangleq (V_B, W_B, V_{LB}, \Theta_B, \rho_B, L_B)$ be two processes in S, then their parallel asynchronous

composition $C \triangleq A \| B$ is given by $C = (V_C, W_C, V_{LC}, \Theta_C, \rho_C, L_C)$, where $V_C = V_A \cup V_B$, $W_C = W_A \cup W_B$, $V_{LC} = V_{LA} \cup V_{LB}$, $\Theta_C = \Theta_A \wedge \Theta_B$, $\rho_C = \rho_A \cup \rho_B$, and $L_C = L_A \wedge L_B$. In case C is closed, i.e., it has no interaction with its environment, we restrict all variables of C to be local variables $V_C = W_C = V_{LC}$.

The closed parameterized system $\mathcal{Q}(N)$ is given by the parallel asynchronous composition of a composable set of N symmetric processes $\mathcal{Q}(N) = P(1) \| P(2) \| ... \| P(N)$. Equivalently, $\mathcal{Q}(N)$ can be defined as a process $(V, V, V, \Theta, \rho, L)$ with $V = \bigcup_{i \in 1..N} V_i$, $\Theta = \bigwedge_{i \in 1..N} \Theta_i$, $\rho = \bigcup_{i \in 1..N} \rho_i$, and $L = \bigwedge_{i \in 1..N} L_i$. All variables of the system are local, since we assume that $\mathcal{Q}(N)$ has no interaction with its environment. A state of the system is an interpretation of all the variables in V. The set of all states is denoted by Σ.

The next state relation ρ_i is defined using a set of atomic actions $A(i)$. Each action α has a precondition (or enable condition) $\text{prec}(\alpha)$, which is a proposition over the variables observable to the process, and an effect part $\text{eff}(\alpha)$, which describes the values of the variables in the next state s', as a function of the current state s. Therefore, α can be described as[1] $\alpha = \text{prec}(\alpha) \wedge \text{eff}(\alpha)$. A state pair $(s, s') \in \rho$, if and only if there exists $i \in 1..N$ and $\alpha \in A(i)$, such that $\text{prec}(\alpha)$ is true for s and the pair of states (s, s') satisfies $\text{eff}(\alpha)$. For all states $s \in \Sigma$, (s, s) belongs to ρ. Our model is equivalent to the interleaving semantics, as only one process executes an action during each transition.

[1]Actions also contain conjuncts of the form $m' = m$ for each variable m of V that must remain unchanged. Therefore, the effect of an action may be considered as the conjunct of $\epsilon(\alpha) \wedge \text{unch}(\alpha)$. In the last formula $\epsilon(\alpha)$ is a boolean combination of predicates of the form $m' = g(s)$, and $\text{unch}(\alpha)$ is the conjunction of predicates of the form $m' = m$. We say that an action "reads" a variable n, when n appears in the expression $g(s)$ of a predicate $m' = g(s)$ of $\epsilon(\alpha)$. An action "modifies" or "writes" a variable m, when there is a predicate $m' = g(s)$ in $\epsilon(\alpha)$ and $g(s) \neq m$. This classification is based on the syntax and can be performed by static analysis.

We assume that all actions can be expressed in disjunctive normal form (dnf), i.e., as a disjunction, in which each disjunct is a conjunction of propositions (non quantified atomic formulas) over variables in V and V'. All next state variables, i.e., variables of V', should appear in exactly one proposition of each disjunct, and be the left hand side expression of that proposition. Moreover, the propositions with next state variables should be equality relations.

The liveness property L_i is a restriction imposed on the infinite behaviors of the system. It can include the conjunction of strong and weak fairness properties specified on some of the actions in $A(i)$. We use $\mathcal{W}_i$ and $\mathcal{S}_i$ to represent the sets of actions with weak and strong fairness properties respectively. Then $L_i \rightarrow \bigwedge_{\alpha \in \mathcal{W}_i} wf(\alpha) \wedge \bigwedge_{\alpha \in \mathcal{S}_i} sf(\alpha)$. The weak and strong fairness properties are defined as

$$wf(\alpha) \triangleq (\Box\Diamond\neg\text{prec}(\alpha)) \vee (\Box\Diamond\langle\text{eff}(\alpha)\rangle)$$

$$sf(\alpha) \triangleq (\Diamond\Box\neg\text{prec}(\alpha)) \vee (\Box\Diamond\langle\text{eff}(\alpha)\rangle)$$

The expression $\langle\text{eff}(\alpha)\rangle$ evaluates to true when action α is executed and the system's state changes [54]. Therefore, for a pair of states (s, s'), it holds

$$(s, s') \models \langle\text{eff}(\alpha)\rangle \Leftrightarrow (s, s') \models \text{eff}(\alpha) \wedge s' \neq s \tag{7.1}$$

We denote as $\alpha\lfloor \texttt{var}_1 \leftarrow \texttt{var}_2, \texttt{var}_3 \leftarrow \texttt{var}_4, ..., \texttt{var}_k \leftarrow \texttt{var}_{k+1} \rfloor$ the predicate obtained from α by replacing each occurrence of the variables $\texttt{var}_1$, $\texttt{var}_3$, ..., $\texttt{var}_k$ with $\texttt{var}_2$, $\texttt{var}_4$, ..., $\texttt{var}_{k+1}$. We will use this notation even if some of the variables $\texttt{var}_1$, $\texttt{var}_3$, ..., $\texttt{var}_k$ do

not appear in α. If none of these variables appear in α, then α remains unchanged after this operation.

A sequence $\sigma = s_0, s_1, ...$ of states, with $\sigma \in \Sigma^\omega$, is a behavior of $\mathcal{Q}(N)$ if σ satisfies the specification $\mathcal{Q}(N)$. More specifically, it must hold that $s_0 \models \Theta$, $\forall i \geq 0 : (s_i, s_{i+1}) \in \rho$, and $\sigma \models L$.

We classify the variables that may exist in $\mathcal{Q}(N)$ as global variables, communication registers, and local variables. The set V_g of global variables is defined as

$$V_g \triangleq \bigcup_{i \in 1..N} V_i - \bigcup_{i \in 1..N} W_i$$

These are the variables that are not owned by any process and, therefore, they are the equivalent of multi-reader, multi-writer variables. We require that the size of this set is finite and independent of the number of processes in the system. The set $V_{\texttt{cr}}$ of variables represents the set of communication registers and is defined as

$$V_{\texttt{cr}} \triangleq \bigcup_{i \in 1..N} \left(W_i - V_{L(i)}\right)$$

Each communication register is a single-writer, multi-reader variable. We require that there is at most one communication register per process and we reserve the name $\texttt{cr}[i]$ for the communication of process $P(i)$, in case such a variable exists[2]. Finally, the set V_L of local variables in the system is given by

$$V_L = \bigcup_{i \in 1..N} V_{L(i)}$$

[2]The extension to a fixed number of communication registers is straightforward.

Symbol	Definition
$\texttt{cr}[j]$	the communication register of process $P(j)$; only process $P(j)$ can modify this variable but all processes can read it
$D_{\texttt{var}}$	the domain of variable var
$\text{eff}(\alpha)$	the effect of an action α; it defines the next state values of the system variables
H	a state predicate expressed over the variables in $V_g \cup \bigcup_{i \in 1..N} V_{L(i)} \cup \{\texttt{cr}[i] \mid i \in 1..N\}$; the correctness property we target is of the form $\Diamond\Box H \rightarrow \Diamond\Box J$
I_H	the network invariant generated during control abstraction
J	a state predicate expressed over the variables in $V_g \cup V_{L(1)} \cup \{\texttt{cr}[1]\}$; the correctness property we target is of the form $\Diamond\Box H \rightarrow \Diamond\Box J$
L_i	the liveness condition of a process i; evaluated only on infinite sequences
$\texttt{lcr}[j]$	the local variable of process $P(j)$ that is a copy of its communication register; these variables are added after the first step of the technique
N	the number of processes in the system
$P(j)$	a process with id = j instantiated from the generic process $P(id)$
$\text{prec}(\alpha)$	the precondition or enable condition of an action α
$\mathcal{Q}(N)$	the parameterized system with N processes
$\tilde{\mathcal{Q}}(N)$	the system obtained after applying the first step of the abstraction technique
$\breve{\mathcal{Q}}(N)$	the system obtained after applying the second step of the abstraction technique
$\mathcal{S}$	the set of actions with strong fairness conditions
$sf(\alpha)$	the strong fairness condition of action α
V_g	the fixed set of global variables that any process can read and modify
$V_{\texttt{cr}}$	$\triangleq \{\texttt{cr}[j] \mid j \in 1..N\}$ the set of communication registers; each communication register is a single-writer multi-reader variable
$V_{L(i)}$	the set of local variables of process $P(i)$, which no process other than $P(i)$ can read or modify
V_i	the set of all variables process $P(i)$ can read or modify
W_i	the set of owned variables of process $P(i)$
$\mathcal{W}$	set of actions with weak fairness conditions
$wf(\alpha)$	weak fairness condition of action α
Θ_i	the initial condition of process $P(i)$
ρ	next state relation of a system; $(s_1, s_2) \in \rho \Leftrightarrow (\exists a \in A : (s_1, s_2) \models a)$, where A is the set of actions
$\Pi_V(s)$	the projection of the state s on the set of variables (or variable) V
Σ	the set of states of the system
φ	the correctness property; for self-stabilizing systems it is equal to $\Diamond\Box H \rightarrow \Diamond\Box J$

Table 7.1. Definition of the symbols used in this chapter.

These are single-writer, single-reader variables. We require that the size of the set $V_{L(i)}$ is finite and independent of N. However, the size of V_L depends on the number of processes

in the system. We call the property that no process $P(j)$ can read or write a variable in $V_{L(i)}$ for $i \neq j$, the *locality restriction*. For a process $P(i)$ all variables that can be read or written and do not belong to $V_{L(i)}$ are the *observable variables*. These variables can be observed by the environment of the process and are used for the communication between the process and its environment, which includes other processes.

If one action α can be obtained from another action β of the same process by replacing any appearance of one communication register $\texttt{cr}[k]$ with another communication register $\texttt{cr}[j]$, then α and β are called syntactically equivalent. A formal description of this relation is given in Section 7.5.2.

We now present our assumptions for the systems we consider. Then we elaborate on the reasons for making these assumptions and their implications. Note that these assumptions may not be necessary for each step of our technique. In the section describing each step we mention the sufficient conditions for soundness and completeness. However, restrictions $\Lambda 1$ – $\Lambda 3$ characterize the systems to which we want to apply all steps of our technique.

$\Lambda 1$. Actions can either read or write at most one communication register in each atomic step.

$\Lambda 2$. The preconditions of the actions do not depend on the values of the communication registers. Therefore, reading a communication register can only be done by the effect part of an action.

$\Lambda 3$. There is no pair of actions that are not syntactically equivalent and have the same effect in a state. More specifically, if $\alpha \in \mathcal{W}_i \cup \mathcal{S}_i$, then for any action

$\beta \in A(i)$ with $\beta \neq \alpha$

$$\forall (s, s') \in \rho : (s, s') \not\models (\langle \text{eff}(\alpha) \rangle \wedge \langle \text{eff}(\beta) \rangle)$$

We believe that the above constraints are common among many applications. Restriction $\Lambda 1$ specifies the low-atomicity constraint. The restriction $\Lambda 2$ has been used in other works ([61],Chapter 9). The reason for this restriction is that reading a communication register is a more expensive operation than reading a local variable and, therefore, should be an atomic action. The decision of a process to execute an action should be based on local variables only. Consequently, communication registers should be copied to local variables before their value is used in the precondition of an action. The intuition behind $\Lambda 3$ is that any transition (s, s') other than the stuttering step can be caused by only one action. However, syntactically equivalent actions are not restricted by $\Lambda 3$. Most systems with a program counter for each process satisfy the $\Lambda 3$ restriction. More specifically, if each instruction has a different successor, the effect of each action of one process is distinct. Since the program counter is a local variable of each process, the effect of each action cannot be simulated by an action of a different process. The restrictions $\Lambda 1$ – $\Lambda 3$ do not need to hold for the fixed set of global variables in V_g.

The above restrictions are needed to enable the abstraction of the communication registers by one global variable using syntactic transformations (Section 7.5). Because of restrictions $\Lambda 1$ and $\Lambda 2$ and the interleaving semantics the value of only one communication register is important in any state. Restrictions $\Lambda 2$ and $\Lambda 3$ make it possible to maintain fairness conditions after the syntactic transformations.

We assume that the correctness property is given in the form $\Diamond\Box H \rightarrow \Diamond\Box J$. This type of condition is very common as a subgoal for self-stabilizing systems. For these systems it usually states that once the environment of a process satisfies a specific persistence condition ($\Diamond\Box H$), the process must satisfy a persistence condition ($\Diamond\Box J$), which is independent of the number of processes in the system. We consider process 1 to be the special process that must satisfy $\Diamond\Box J$. Therefore, J is expressed over the variables in $V_g \cup W_i$.

In this section we presented the assumptions for the systems we consider and the notation we use. Table 7.1 displays some commonly used symbols and their definitions. In the next section we give an overview of the proposed technique for the verification of these systems.

7.4. Overview of our Approach

In this section we give an overview of the proposed abstraction technique for checking the correctness of parameterized self-stabilizing systems.

0. *Preprocessing*: Transform the system to a closed system of N processes, which is amenable to our approach.
1. *Reducing the observable state space*: Abstract the set of observable variables of each process to a fixed finite set, if the size of the observable state space depends on N (Section 7.5).
2. *Simplifying the correctness property*: Simplify the correctness property from $\Diamond\Box H \rightarrow \Diamond\Box J$ to $\Diamond\Box J$, by transforming the system to a system that satisfies $\Box H$ (Section 7.6).

3. *Constructing a network invariant*: Generate a process I that is a network invariant for $\mathcal{Q}(N)$ (Section 7.7).
4. *Model-checking*: Verify that $(P(1)\|I) \models \Diamond\Box J$ by model-checking. If no counterexample is found, then for all $N : \mathcal{Q}(N) \models (\Diamond\Box H \rightarrow \Diamond\Box J)$.

Some self-stabilizing systems do not satisfy the assumptions made in Section 7.3. For example, if the domains of some variables are not finite or if there exist relations specified over the ids of the processes, the preprocessing step makes the system amenable to our technique. During the preprocessing step data abstraction [44] or other abstraction techniques can be employed.

For systems amenable to our technique the size of the observable state space of each process may depend on N. In those cases, control abstraction is not directly applicable. During the *Reducing the observable state space* step, the set of observable variables of each process is abstracted to a fixed finite set. This happens by making all communication registers owned by $P(2), ..., P(N)$ local variables and allowing the communication to occur through a fixed set of new global variables. The communication actions of the system and the liveness conditions have to be modified, as well. The purpose of this step is to enable the step of control abstraction.

In the next step the correctness property is simplified from $\Diamond\Box H \rightarrow \Diamond\Box J$ to $\Diamond\Box J$ by transforming the system to a system that preserves $\Box H$. This a sound and complete transformation for self-stabilizing systems. After the transformation the correctness property, i.e., $\Diamond\Box J$, is expressed over the variables in $V_g \cup V_{L(1)}$ enabling the abstraction of the set of variables $\bigcup_{i \in 2..N} V_{L(i)}$.

The next step is to generate the network invariant. The generation of the actions is based on syntax analysis of the distributed algorithm and on the property H. After generating the network invariant I, the last step is to use model-checking on the system composed of process $P(1)$ and I. If $(P(1)\|I)$ satisfies the simplified property $\Diamond\Box J$, then the parameterized $\mathcal{Q}(N)$ system satisfies $\Diamond\Box H \rightarrow \Diamond\Box J$ for all N. The last step can be performed automatically by any model-checking tool.

We assume that the preprocessing step, if needed, has already been applied to the input algorithms. Hence, we focus on the *Reducing the observable state space*, *Simplifying the correctness property*, and *Constructing a network invariant* steps.

7.5. Reducing the Observable State Space

In this section we describe the first step of the proposed abstraction technique. The goal of this step is to reduce the number of observable variables of each process to a fixed finite set. This reduction enables the method of network invariants, i.e., control abstraction, to abstract the state space of an arbitrary number of processes.

For the soundness of the abstraction method described in this step the system does not have to be self-stabilizing. However, it needs to satisfy conditions $\Lambda 1$ – $\Lambda 3$. Moreover, the correctness property φ must be an LTL property in which the only temporal operators that can occur are $\Box$ and $\Diamond$. Note that self-stabilizing systems comply with this requirement (Section 7.2).

Let $V_{\mathtt{cr}}^{2..N}$ be the set of the communication registers, whose owners are the processes $P(2) - P(N)$, i.e.,

$$V_{\mathtt{cr}}^{2..N} \triangleq \{\mathtt{cr}[j] \mid j \in 2..N\}$$

The purpose of this step is to replace these variables with a global variable $\texttt{cr}_a[2]$ and a set of $N-1$ local variables, i.e., one local variable $\texttt{lcr}[j]$ for each process $P(j)$ (Figure 7.1). The idea behind this transformation is that because of the low-atomicity constraint ($\Lambda 1$) and the interleaving semantics at most one variable of $V_{\texttt{cr}}^{2..N}$ can be read or written during each transition of the system. Therefore, before the action that causes the transition is executed, only one communication register and its value are important. Hence, instead of $N-1$ communication registers, we use only one global variable ($\texttt{cr}_a[2]$) and give each process $P(j)$, $j \in 2..N$, the ability to copy the value stored in $\texttt{lcr}[j]$ to $\texttt{cr}_a[2]$. The variable $\texttt{lcr}[j]$ stores the value that the communication register $\texttt{cr}[j]$ would have in the original system. In the next section we formally define the transformation and prove its soundness.

7.5.1. Obtaining the Abstract System

We denote the system produced during this step as $\tilde{\mathcal{Q}}(N)$. System $\tilde{\mathcal{Q}}(N)$ is defined by a composition of a number of processes. We show how the specification of the special process $\tilde{P}_1$ and the generic process $\tilde{P}(i)$ can be obtained.

The new specification $\tilde{P}_1$ is given by $\tilde{P}_1 = (\tilde{V}_1, W_1, V_{L(1)}, \Theta_1, \tilde{\rho}_1, \tilde{L}_1)$, where

$\tilde{V}_1$: The set $\tilde{V}_1$ of variables is obtained by removing from V_1 all communication registers of processes $P(2), ..., P(N)$ and adding the new variable $\texttt{cr}_a[2]$. More formally,

$$\tilde{V}_1 \triangleq V_1 - V_{\texttt{cr}}^{2..N} \cup \{\texttt{cr}_a[2]\}$$

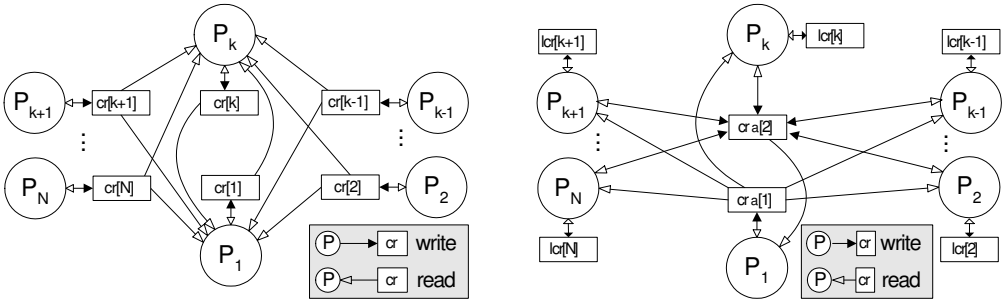


Figure 7.1. Left is the process graph before the transformation. Each process writes to its own communication register and reads the communication registers of all processes. For clarity the read edges only of $P(1)$ and $P(k)$ are shown. The right part represents the system after the transformation. The new variable $\mathtt{cr}_a[2]$ is a global variable. It is a multi-reader, multi-writer variable. The new local variables `lcr` are shown next to each process.

$\tilde{\rho}_1$: The new next state relation $\tilde{\rho}_1$ is given by a new set $\tilde{A}_1$ of actions. For each action of the original set $A(1)$ a new action is included in $\tilde{A}_1$. The new action is obtained by replacing with $\mathtt{cr}_a[2]$ any occurrence of a communication register $\mathtt{cr}[j]$ where $j \in 2..N$. More formally,

$$\tilde{A}_1 \triangleq \{\alpha\lfloor \mathtt{cr}[j] \leftarrow \mathtt{cr}_a[2]\rfloor \mid \alpha \in A(1), j \in 2..N\}$$

There are no actions of $P(1)$ that modify communication registers of $V_{\mathtt{cr}}^{2..N}$.

$\tilde{L}_1$: There are three kinds of liveness constraints that the transformed process has; weak, strong, and constant fairness constraints. Each action $\tilde{\alpha}$ of the new system, which can be obtained by any element of a subset A_s of $A(1)$, inherits the strongest among the fairness conditions specified on the actions of A_s. Let $\tilde{\mathcal{W}}_1$ and $\tilde{\mathcal{S}}_1$ be the subsets of $\tilde{A}_1$ that contain the actions with weak and strong fairness

constraints, respectively. Then we have

$$\begin{aligned}\tilde{\mathcal{S}}_1 &\triangleq \{\alpha\lfloor \mathtt{cr}[j] \leftarrow \mathtt{cr}_a[2]\rfloor \mid \alpha \in \mathcal{S}_1, j \in 2..N\} \\ \tilde{\mathcal{W}}_1 &\triangleq \{\alpha\lfloor \mathtt{cr}[j] \leftarrow \mathtt{cr}_a[2]\rfloor \mid \alpha \in \mathcal{W}_1, j \in 2..N\} \; - \; \tilde{\mathcal{S}}_1\end{aligned}$$

where $\mathcal{W}_1$ and $\mathcal{S}_1$ are the sets of actions of $A(1)$ with weak and strong fairness conditions.

Besides the strong and weak fairness conditions on actions, liveness conditions related to constants are specified. Let FS be the finite domain of each communication register. Then suppose that for any N and for all behaviors of $\mathcal{Q}(N)$, there exists some $k \in 2..N$ and $v_k \in$ FS, such that it holds $\Box(\mathtt{cr}[k] = v_k)$. If there exists an action $\alpha \in \mathcal{W}_1$, reading $\mathtt{cr}[k]$, we define condition $c(\alpha)$ obtained from eff(α) by replacing each occurrence of $\mathtt{cr}[k]$ with the value v_k, i.e.,

$$c(\alpha) \triangleq \text{eff}(\alpha)\lfloor \mathtt{cr}[k] \leftarrow v_k\rfloor$$

We define constraint

$$cf(\alpha) \triangleq \Box\Diamond\neg\text{prec}(\alpha) \vee \Box\Diamond\langle c(a)\rangle$$

For an action $\alpha \in \mathcal{S}_1$ accessing $\mathtt{cr}[k]$, the corresponding constraint will be

$$cf(\alpha) \triangleq \Diamond\Box\neg\text{prec}(\alpha) \vee \Box\Diamond\langle c(a)\rangle$$

Note that index k does not need to be the same for all behaviors. If the fairness properties are specified on a set of syntactically equivalent read actions that are

defined for all $i \in 2..N$, the existence of a constant value in $V_{\mathtt{cr}}^{2..N}$ for all behaviors of $\mathcal{Q}(N)$ is sufficient for creating the constraint. We denote as $\mathcal{C}_1$ the set of the actions from which constant fairness conditions are generated. Then $\tilde{L}_1$ can be expressed as

$$\tilde{L}_1 = \bigwedge_{\tilde{\alpha} \in \tilde{\mathcal{W}}_1} wf(\tilde{\alpha}) \ \wedge \ \bigwedge_{\tilde{\alpha} \in \tilde{\mathcal{S}}_1} sf(\tilde{\alpha}) \ \wedge \ \bigwedge_{\alpha \in \mathcal{C}_1} cf(\alpha)$$

For convenience we rename the communication register of $\tilde{P}_1$ as $\mathtt{cr}_a[1]$. The sets of local and owned variables and the initial condition remain the same.

Similarly, the generic process $\tilde{P}(i) = (\tilde{V}_i, \tilde{W}_i, \tilde{V}_{L(i)}, \tilde{\Theta}_i, \tilde{\rho}_i, \tilde{L}_i)$ is defined.

$\tilde{V}_i$: The set $\tilde{V}_i$ is obtained by removing the communication registers $\mathtt{cr}[2], ..., \mathtt{cr}[N]$ and adding the new global variable $\mathtt{cr}_a[2]$[3] and the local variable $\mathtt{lcr}[i]$. Formally,

$$\tilde{V}_i \triangleq V_i - V_{\mathtt{cr}}^{2..N} \cup \{\mathtt{cr}_a[2], \mathtt{lcr}[i]\}$$

$\tilde{W}_i$: The set of owned variables of each process becomes equal to the set of local variables. This is because the only variable of W_i in the original system that can be read by another process is the communication register $\mathtt{cr}[i]$. Since this variable is removed, it holds $\tilde{W}_i = \tilde{V}_{L(i)}$.

$\tilde{V}_{L(i)}$: To the set $\tilde{V}_{L(i)}$ of local variables the variable $\mathtt{lcr}[i]$ is added, i.e.,

$$\tilde{V}_{L(i)} \triangleq V_{L(i)} \cup \{\mathtt{lcr}[i]\}$$

[3]Variable $\mathtt{cr}[1]$ is renamed to $\mathtt{cr}_a[1]$ but only for naming convenience.

$\tilde{\Theta}_i$: The predicate $\tilde{\Theta}_i$ is generated from Θ_i by replacing each occurrence of $\texttt{cr}[i]$ with $\texttt{lcr}[i]$, i.e.,

$$\tilde{\Theta}_i \triangleq \Theta\lfloor \texttt{cr}[i] \leftarrow \texttt{lcr}[i] \rfloor$$

$\tilde{\rho}_i$: The next state relation $\tilde{\rho}_i$ is defined by a new set of actions $\tilde{A}(i)$ obtained from $A(i)$. For each action of $A(i)$ we create an action in $\tilde{A}(i)$ by replacing any occurrence of $\texttt{cr}[j] \in V_{\texttt{cr}}^{2..N}$ with $\texttt{cr}_a[2]$. Set $A(i)$ may also contain actions that write a value to $\texttt{cr}[i]$. We replace each conjunct $\texttt{cr}[i]' = \epsilon(\alpha)$ of a disjunct of an action α, where $\epsilon(\alpha) \neq \texttt{cr}[i]$, with two conjuncts that specify the values of $\texttt{cr}_a[2]$ and $\texttt{lcr}[i]$ in the next state, i.e., $\texttt{cr}_a[2]' = \epsilon(\alpha)$ and $\texttt{lcr}[i]' = \epsilon(\alpha)$. A conjunct of the form $\texttt{cr}[i]' = \texttt{cr}[i]$ is replaced with $\texttt{cr}_a[2]' = \texttt{cr}_a[2]$ and $\texttt{lcr}[i]' = \texttt{lcr}[i]$.

Moreover, an action $\tilde{\alpha}_i^0$ is added that is always enabled and modifies $\texttt{cr}_a[2]$ to $\texttt{lcr}[i]$ leaving all other variables unchanged, i.e.,

$$\tilde{\alpha}_i^0 \triangleq \quad \wedge \quad \texttt{cr}_a[2]' = \texttt{lcr}[i] \qquad \bigwedge_{\texttt{var} \in V - \{\texttt{cr}_a[2]\}} \texttt{var}' = \texttt{var}$$

$\tilde{L}_i$: The weak, strong, and constant fairness properties of the new specification are built the same way as $\tilde{L}_1$ was built from L_1. Namely, each action $\tilde{\alpha}$ inherits the strongest fairness constraint among the constraints of the actions that can be used to generate $\tilde{\alpha}$. Constant fairness constraints are added if there exist constant values that the generic process must read based on fairness properties. The action $\tilde{\alpha}_i^0$ does not have any fairness constraint.

An additional initial condition is added to the new system restricting the variable $\texttt{cr}_a[2]$ to have the same value as one of the registers $\texttt{lcr}[j]$ for $j \in 2..N$. In our formulation to include the new initial predicate θ_g, we define a new process $P_{init} \triangleq (V, \emptyset, \emptyset, \theta_g, \emptyset, \text{True})$ with $\theta_g \triangleq \exists j \in 2..N : \texttt{lcr}[j] = \texttt{cr}_a[2]$. This new process has no actions, owned variables, or liveness conditions and, therefore, does not specify any temporal property other than an assertion for the initial state. The transformed system after the application of this step is given by

$$\tilde{\mathcal{Q}}(N) = P_{init} \| \tilde{P}_1 \| \tilde{P}(2) \| ... \| \tilde{P}(N)$$

where $\tilde{P}(2), ..., \tilde{P}(N)$ are instantiations of the generic process $\tilde{P}(i)$. Alternatively, we consider $\tilde{\mathcal{Q}}(N)$ as a closed process $\tilde{\mathcal{Q}}(N) = (\tilde{V}, \tilde{V}, \tilde{V}, \tilde{\Theta}, \tilde{\rho}, \tilde{L})$, where each component is defined by the composition of the process $P_{init}, \tilde{P}_1, \tilde{P}(2), ..., \tilde{P}(N)$.

If the correctness property of the system is expressed over variables in $V_{\texttt{cr}}^{2..N}$, these variables are replaced with the corresponding local copies $\texttt{lcr}$. Formally, the new correctness property $\tilde{\varphi}$ is defined as

$$\tilde{\varphi} \triangleq \varphi \lfloor \texttt{cr}[2] \leftarrow \texttt{lcr}[2], ..., \texttt{cr}[N] \leftarrow \texttt{lcr}[N] \rfloor$$

The following theorem states that the abstraction technique is sound.

Theorem 7.1. *If for $N \in \mathbb{N}$ it holds that $\tilde{\mathcal{Q}}(N) \models \tilde{\varphi}$, then $\mathcal{Q}(N) \models \varphi$.*

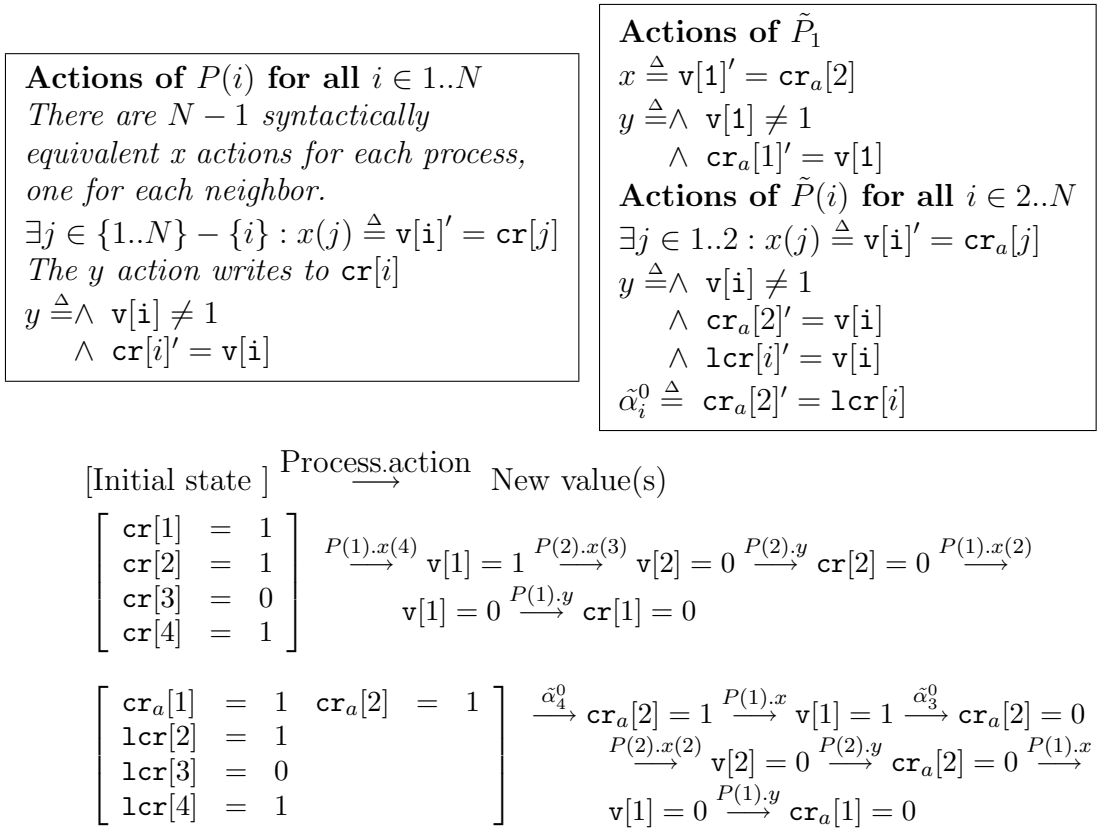


Figure 7.2. This is an example of the application of the transformation. Upper left figure shows the actions of $\mathcal{Q}(N)$. Upper right figure shows the actions of $\tilde{\mathcal{Q}}(N)$. For simplicity the part of each action specifying the variables left unchanged is not shown. The upper sequence is part of a behavior of the original system. The bottom sequence is the corresponding behavior segment of the abstract system. The projections of two behavior segments over the variables in $V - V_{\mathtt{cr}}^{2..N}$ are stuttering equivalent. Only the relevant values are displayed in the figure.

Proof. The proof is based on the theory of refinement mappings [1]. We augment system $\mathcal{Q}(N)$ with a prophecy variable[4] $\pi \in 2..N$ to obtain system $\mathcal{Q}(N)^\pi$. The prophecy variable π holds the index of the next communication register in $V_{\mathtt{cr}}^{2..N}$ that will be read

[4]A history variable is a variable that records past information and does not affect the behavior of the system. A prophecy variable is similar to a history variable, but instead of recording past information, it predicts future information.

or written. The initial value of π can be any element of $2..N$. We add one more action α_π that is always enabled and changes π to any of the values in $2..N$. Action α_π leaves all other variables of the system unchanged. All other actions of the new system do not modify π. Actions that read or modify variable $\texttt{cr}[j] \in V_{\texttt{cr}}^{2..N}$ are guarded by condition $\pi = j$. The new system's liveness condition is the temporal formula L which is the liveness condition of $\mathcal{Q}(N)$. It is easy to prove then that $\mathcal{Q}(N)^\pi$ is a system obtained from $\mathcal{Q}(N)$ by adding a prophecy variable.

We define a function f from the state space of $\mathcal{Q}(N)^\pi$ to the state space of $\tilde{\mathcal{Q}}(N)$ and show that it is a refinement mapping. Let $s_c = (e, \texttt{cr}[1], \texttt{cr}[2], ..., \texttt{cr}[N], \pi)$ be a state of $\mathcal{Q}(N)^\pi$, where $e \in s_c[V - V_{\texttt{cr}} - \{\pi\}]$. Then $f(s_c) = (e, \texttt{cr}_a[1], \texttt{cr}_a[2], \texttt{lcr}[2], \texttt{lcr}[3], ..., \texttt{lcr}[N])$, where $\texttt{cr}_a[1] = \texttt{cr}[1]$, $\texttt{lcr}[2] = \texttt{cr}[2]$, ..., $\texttt{lcr}[N] = \texttt{cr}[N]$ and $\texttt{cr}_a[2] = \texttt{cr}[\pi]$. In order for f to be a valid refinement mapping the following conditions need to be satisfied [1]:

R1. For each s_c in the state space of $\mathcal{Q}(N)^\pi$:

(a) $\Pi_{\tilde{V} - V_{\texttt{lcr}}^{2..N} - \{\texttt{cr}_a[2]\}}(f(s_c)) = \Pi_{V - V_{\texttt{cr}}^{2..N} - \{\pi\}}(s_c)$

(b) $\Pi_{V_{\texttt{lcr}}^{2..N}}(f(s_c)) = \Pi_{V_{\texttt{cr}}^{2..N}}(s_c)$

where the operator $\Pi_V(s)$ denotes the projection of the state s over the set V of variables and set $V_{\texttt{lcr}}^{2..N}$ is the set of local variables $\{\texttt{lcr}[j] \mid j \in 2..N\}$.

R2. $f(F^\pi) \subseteq F_a$, where F^π is the set of initial states of $\mathcal{Q}(N)^\pi$ and $\tilde{F}$ the set of initial states of $\tilde{\mathcal{Q}}(N)$.

R3. If $(s_c, t_c) \in \rho^\pi$ then $(f(s_c), f(t_c)) \in \tilde{\rho}$ or $f(s_c) = f(t_c)$, where ρ^π is the next state relation of $\mathcal{Q}(N)^\pi$.

R4. $f(\mathcal{X}^\pi) \subseteq \tilde{L}$, where $\mathcal{X}^\pi$ is the set of computations specified by $\mathcal{Q}(N)^\pi$.

Properties R1 and R2 hold by construction of $\mathcal{Q}(N)^\pi$ and $\tilde{\mathcal{Q}}(N)$. From R1 it follows that all variables over which φ is expressed have an 1-to-1 mapping to variables of $\tilde{\mathcal{Q}}(N)$ over which $\tilde{\varphi}$ is expressed. For R3 we can show that the transition caused by an action β of $\mathcal{Q}(N)^\pi$, which originated from an action α of the original system, can be simulated in the abstract system by a transition using $\tilde{\alpha}$, which is the action created from α by the procedure described above. A transition caused by α_π can be simulated by the transition of an action $\tilde{\alpha}_j^0$ in the abstract system. Property R4 is the hardest to prove. Let $f(\sigma^\pi) \not\models L_a$ be a sequence, such that $\sigma^\pi \models \mathcal{Q}(N)^\pi$. We show that this leads to a contradiction. Sequence $f(\sigma^\pi)$ must violate a weak, strong, or constant fairness property of $\tilde{L}$. Assume $f(\sigma^\pi)$ violates the weak fairness property of action $\tilde{\alpha}$. Then there exists action α in the original system, from which $\tilde{\alpha}$ is obtained, such that $\sigma^\pi \models \Box\Diamond\neg\text{prec}(\alpha) \vee \Box\Diamond\langle\text{eff}(\alpha)\rangle$. If $\sigma^\pi \models \Box\Diamond\neg\text{prec}(\alpha)$, then we know $f(\sigma^\pi) \models \Box\Diamond\neg\text{prec}(\tilde{\alpha})$, since α and $\tilde{\alpha}$ have the same precondition which is expressed over variables in $V - V_{\texttt{cr}}$ only (assumption $\Lambda 2$). Therefore, it must hold that $\sigma^\pi \models \Box\Diamond\langle\text{eff}(\alpha)\rangle$. If action α does not read or modify a variable in $V_{\texttt{cr}}$, we know that $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\tilde{\alpha})\rangle$. Consequently, α must read or modify a communication register. Then using assumption $\Lambda 3$, we can show that $\langle\text{eff}(\alpha)\rangle$ is infinitely often satisfied because α or a syntactically equivalent action is executed infinitely often. In either case we have that $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\tilde{\alpha})\rangle$, which leads to a contradiction.

For example, if α reads a variable $\texttt{cr}[j]$, then $\sigma^\pi \models \Box\Diamond\langle\text{eff}(\alpha)\lfloor\texttt{cr}[j] \leftarrow \texttt{cr}[\pi]\rfloor\rangle$. This implies that $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\alpha)\lfloor\texttt{cr}[j] \leftarrow \texttt{cr}_a[2]\rfloor\rangle$, or, equivalently, $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\tilde{\alpha})\rangle$.

In a similar way, we can prove that strong and constant fairness conditions of $\tilde{L}$ are satisfied by any sequence $f(\sigma^\pi)$ with σ^π a behavior of $\mathcal{Q}(N)^\pi$. $\Box$

In case $\mathcal{Q}(N)$ is a self-stabilizing system, system $\tilde{\mathcal{Q}}(N)$ is not only a sound abstraction of $\mathcal{Q}(N)$, but has the additional property that every reachable state is an initial state.

Lemma 7.1. *If the system $\mathcal{Q}(N)$ is self-stabilizing, every reachable state of system $\tilde{\mathcal{Q}}(N)$ is an initial state.*

Proof. We show that $\tilde{\Theta}$ is an invariant of $\tilde{\mathcal{Q}}(N)$. If $\mathcal{Q}(N)$ is self-stabilizing, then Θ_i equals[5] True for all $i \in 1..N$. Therefore, for system $\tilde{\mathcal{Q}}(N)$ we have $\tilde{\Theta} = \theta_g$. Every action $\tilde{\alpha}$ of $\tilde{A}(i)$ that changes the value of $\mathtt{cr}_a[2]$ either assigns the same value to $\mathtt{lcr}[i]$ or copies the value of $\mathtt{lcr}[i]$ to $\mathtt{cr}_a[2]$. Actions of $\tilde{A}_1$ do not modify $\mathtt{cr}_a[2]$. Hence, θ_g is preserved by all actions of $\tilde{\mathcal{Q}}(N)$. Predicate θ_g holds initially and is an invariant of $\tilde{\mathcal{Q}}(N)$. Consequently, all reachable states of $\tilde{\mathcal{Q}}(N)$ are initial states. □

If φ is expressed in the form $\Box\Diamond H \rightarrow \Box\Diamond J$, as it is expected for self-stabilizing systems, then only H needs to modified for $\tilde{\varphi}$ to be obtained. This is because variables $\mathtt{cr}[2], ..., \mathtt{cr}[N]$ do not appear in J, which is expressed over the owned variables W_1 of the special process $P(1)$. The new correctness property is denoted as $\Diamond\Box\tilde{H} \rightarrow \Diamond\Box J$.

Next section describes sufficient conditions for the completeness of the abstraction technique.

7.5.2. General Conditions for Completeness of the Abstraction Technique

In this subsection we present the conditions under which this step of the abstraction technique is complete.

[5]In some languages predicate Θ_i may specify the domain of each variable owned by $P(i)$. The proof holds in this case, as well.

An important concept is that of syntactically equivalent actions. Suppose α and β are actions of $\mathcal{Q}(N)$. Then $\alpha \equiv_{se} \beta$ if and only if α and β belong to the same process and β can be obtained from α by replacing every instance of $\texttt{cr}[j]$ with $\texttt{cr}[k]$, where j and k are constants in $2..N$. The relation $\equiv_{se}$ is an equivalence relation. All actions of the same equivalence class are transformed to one action by our technique.

The following conditions are sufficient for completeness

C1 Every reachable state of the system is an initial state.

C2 The read actions of the system form equivalence classes, such that the size of each equivalence class is not bounded from above by a constant as N increases.

C3 The number of processes in the system is not bounded from above.

C4 For all N_1 and N_2 with $N_1 < N_2$, the systems $\mathcal{Q}(N_1)$ and $\mathcal{Q}(N_2)$ have the same formula as liveness constraint.

C5 No read action has a fairness constraint.

If $\mathcal{Q}(N)$ is a self-stabilizing system, then C1 is always true. Conditions C2 and C3 are commonly satisfied by uniform parameterized systems, in which no process distinguishes a finite number of its neighbors as special processes. Distributed algorithms that read the values of all neighbors in a loop, which is not an atomic action, satisfy condition C5. Finally, in many cases condition C4 is satisfied as well. This is because the environment of process 1 is normally not constrained by fairness conditions. A fairness constraint on the environment, i.e., an expectation that the environment at some point in the future will execute an action under some conditions, can sometimes be added to the condition $\Diamond\Box H$ of the original condition property ($\Diamond\Box H \rightarrow \Diamond\Box J$). Therefore, by increasing the convergence steps we can remove some fairness requirements for the environment.

Theorem 7.2. *If C1 – C5 hold and* $\tilde{\mathcal{Q}}(N) \not\models \Diamond\Box\tilde{H} \rightarrow \Diamond\Box J$*, then* $\exists K \geq N : \mathcal{Q}(K) \not\models \Diamond\Box H \rightarrow \Diamond\Box J$*.*

Proof. The idea of the proof is to construct a counterexample for some instance of the concrete system using the counterexample of the abstract system. Since the correctness property is a liveness property, the abstract counterexample is a lasso-shaped sequence, i.e., a cycle and a path leading from an initial state to a state in the cycle. This cycle represents the infinite part of the counterexample. However, because in our case the correctness property is $\Diamond\Box\tilde{H} \rightarrow \Diamond\Box J$, all states of the cycle satisfy $\tilde{H}$ and there is at least one state that satisfies $\neg J$. Moreover, every state is an initial state (C1) and, therefore, we can consider only the cycle in the counterexample, which satisfies $\Diamond\Box\tilde{H} \wedge \Box\Diamond\neg J$. Using this cycle we can produce a cycle in the concrete state space, which is a behavior of $\mathcal{Q}(K)$ for some $K \geq N$.

First, we determine the number of processes K in the concrete system. Then using induction we show how we can create a concrete counterexample from the abstract counterexample.

To determine the number K of processes we consider the reads in the cycle of the abstract counterexample. There could be a read action $\tilde{\alpha}$ performed by a process j in the abstract counterexample, such that for any k in the equivalence class of that action α in the concrete system $\mathtt{cr}_a[2] \neq \mathtt{cr}[k]$, i.e., the value, which $\mathtt{cr}_a[2]$ has, is not equal to a communication register that process j could read by executing action α. Because of condition C3 we can add more processes in the system; and due to C2, we know that by increasing the number of processes, we can add at least one new process m to the equivalence class of the read action of j. Since any reachable state is an initial state and

because the processes are uniform, we choose $\texttt{cr}[m] = \texttt{cr}_a[2]$ and a corresponding valid local state for m. Consequently, we start from a state s_0 in the abstract cycle and for each transition (s_i, s_{i+1}), we determine whether the action $\tilde{\alpha}$ that caused the transition is a read action. In case $\tilde{\alpha}$ is a read action, let $[\alpha]_{se}$ be the equivalence class of actions from which $\tilde{\alpha}$ was obtained. We make sure that there is at least one process j with $\texttt{lcr}[j] = \texttt{cr}_a[2]$ in s_i and one action of $[\alpha]_{se}$ reading $\texttt{cr}[j]$. We repeat these steps until we check all transitions of the cycle. We denote the number of processes in the system after the procedure as K. The added processes do not perform any action and, therefore, their local states and owned variables maintain the same values. Consequently, we still have a cycle in the extended state space. Moreover, because of C4 there are no fairness constraints added that could be violated.

The second step of the proof is to build the concrete counterexample for system $\mathcal{Q}(K)$ from the abstract counterexample. We start again from state s_0. We create state t_0 by assigning to the local variables, i.e, $\bigcup_{i \in 1..K} V_{L(i)}$, of the K processes the same values as in s_0. We do the same for the global variables in V_g. For the communication registers we assign the values of the local copies, i.e., $\forall j \in 1..N : \texttt{cr}[j] = \texttt{lcr}[j]$. For every transition (s_i, s_{i+1}) of the abstract system caused by $\tilde{\alpha}$, we create transition (t_i, t_{i+1}) in the concrete system by executing action α from which $\tilde{\alpha}$ is obtained. If $\tilde{\alpha}$ is an $\tilde{\alpha}^0_j$ action, i.e., it is not obtained from an action in the original system, we execute the stuttering step. The action α determined this way has the same effect on the local and global variables as $\tilde{\alpha}$. For the communication registers the effect is the same as the effect of $\tilde{\alpha}$ on the local copies. The preconditions of the actions do not depend on the communication registers and, therefore, if $\tilde{\alpha}$ is enabled in s_i, α is enabled in t_i.

Since property J is expressed over the variables in $V - V_{\mathtt{cr}}^{2..N}$ and the projections of the two counterexamples on these variables are stuttering equivalent sequences, the concrete cycle satisfies $\Box\Diamond\neg J$. Moreover, the new processes $N+1, ..., K$ added to the system are assigned a state that another process in $2, ..., N$ has. Since there are no relations over ids in H, the concrete cycle satisfies $\Diamond\Box H$, as well. Actions used which do not read any communication register have single corresponding actions in abstract system with the same fairness conditions. Therefore, their fairness conditions are satisfied. Moreover, because of condition C5 the fairness conditions of the read actions cannot be violated. $\Box$

Conditions C1 – C5 are sufficient to prove completeness, but not all of them are necessary. For example, C2 – C4 are not needed if the equivalence classes of the read actions include all elements of $\mathtt{cr}[1], ..., \mathtt{cr}[N]$. In such a case we do not need to extend the number of processes of the system. Conditions C4 and C5 are very restrictive. Therefore, for a specific class of self-stabilizing systems, i.e., silent self-stabilizing systems, we define an alternative set of conditions in the next section.

7.5.3. Silent Self-Stabilizing Systems and Completeness Conditions

For a special class of self-stabilizing systems, which are called silent self-stabilizing systems, we can define less restrictive conditions for completeness. A silent self-stabilizing system is a self-stabilizing system in which after a process has recovered from a fault, its output becomes fixed [27]. In our formulation that means that its communication register will maintain a constant value. Assume that the assertion H of the correctness condition implies that there exists a process $P(k)$ that is silent. Moreover, there is a correction dependency of the special process $P(1)$ on the silent process. More specifically, it is assumed

that after the system satisfies H, the process $P(1)$ will eventually recover after reading the value of $\texttt{cr}[k]$ no matter what other values it may read or actions it may execute. For those systems we can prove that the abstraction method of this step is complete.

First we define the exclusive correction dependency. If the recovery of $P(1)$ depends only on value v_k and is independent of the values of the other communication registers, as long as H is satisfied, we say that the relation between $P(1)$ and $P(k)$ is an *exclusive correction dependency.*

In the case of a silent self-stabilizing algorithm we can replace conditions C4 and C5 with the conditions below.

C4' Condition H implies the existence of a silent process $P(k)$ with $k \in 2..N$ for any N.

C5' The recovery of $P(1)$ is exclusively dependent on $P(k)$.

C6' All read actions of $P(1)$ form one syntactically equivalence class of $N-1$ elements.

C7' The read actions of processes $P(2)$, ..., $P(N)$ do not have fairness constraints.

Conditions C1 – C3 and C4' – C7' are sufficient for the completeness of the approach. In the case of silent self-stabilizing systems, conditions C4' – C7' are less restrictive than C4 and C5.

Theorem 7.3. *If conditions C1 – C3 and C4' – C7' hold, then the abstraction method is complete.*

Proof. Assume that for the abstract system $\tilde{\mathcal{Q}}(N)$ we obtain a counterexample. As in the proof of Theorem 7.2 we are interested only in the cyclic part of the counterexample,

which satisfies $\Box\tilde{H} \wedge \Box\Diamond\neg J$. This is because starting from any state of the cycle, we can build an infinite behavior that is a counterexample.

Because of C6' there is only one read action that reads the $\texttt{cr}[k]$ variable. We call this action β. Let us assume that $\tilde{P}_1$ does not read variable $\texttt{cr}_a[2]$, when $\texttt{cr}_a[2] = v_k$. Since the counterexample is valid for the abstract system, it satisfies all fairness constraints including the constant fairness constraints. We start by assuming that there is no constant fairness constraint created from action β for value v_k. Then there is no fairness constraint on the read action β of $P(1)$ in the original system that guarantees that the register $\texttt{cr}[k]$ is read. Because of that any infinite behavior of $\mathcal{Q}(N)$ that is obtained by never allowing the action β to be executed is a valid counterexample for the system. Therefore, there must be a fairness constraint on β, which implies the existence of a constant fairness constraint in the abstract system. Since value v_k is never read, the constant fairness constraint is satisfied by the precondition of action β. We create a concrete counterexample from the abstract counterexample. We replace every action that does not read $\texttt{cr}_a[2]$ with its corresponding action in the concrete system. For the read actions of $P(1)$ we do not need to increase the number of processes because of the assumption on the equivalence class size (C6'). Moreover, in the case of a read action coming from another process we increase the number of processes, so that the equivalence classes can be extended. If action β has a weak fairness condition in the concrete system, then there is a state s_a in the abstract counterexample, such that $s_a \models \neg\text{prec}(\beta)$. The precondition of β is expressed over variables in $V_{L(1)}$ and, consequently, there is a state s_c in the cycle of the concrete extended state space, such that $s_c \models \neg\text{prec}(\beta)$. Because of that $P(1)$ satisfies all its read

fairness constraints. All other fairness constraints are also satisfied (C7'). Therefore, the concrete counterexample is a valid counterexample.

The last case is when $\tilde{P}_1$ reads the value v_k in the abstract counterexample. We denote as s_a the state before the read and s_b the state after the read. If we try to build an extended system and create a concrete counterexample by taking the corresponding actions of the original system, we may fail. The reason is each action that is syntactically equivalent to β may have a fairness constraint. Since all these actions are transformed to one action in the abstract system with one fairness constraint, the infinitely often execution of that action is sufficient to create a valid abstract counterexample. However, this is not the case in the concrete system, in which all $N-1$ actions of that equivalence class need to executed infinitely often. Let t_a and t_b be the corresponding states of s_a and s_b in the extended concrete state space. We create an execution segment from t_a to a state t_b^{N-1} by executing $N-1$ read actions. This execution segments satisfies the additional fairness requirements. Then by taking the segment from t_b^{N-1} to t_a we create an infinite behavior that is a counterexample in the concrete state space. In state t_a all read actions are enabled. We choose one of those actions, β_j, that reads communication register $\texttt{cr}[j] \neq v_k$ with $j \in 2..N - \{k\}$. If after the execution of β_j there is no finite sequence that can make the read action enabled again then we are done. Otherwise, let the new state in which β is enabled be t_b^1. If in state t_b^1 every execution starting with β satisfies the specification, we reach a contradiction. This is because we assumed that $P(1)$ is dependent exclusively on value v_k of variable $\texttt{cr}[k]$. Therefore, there still exists an infinite counterexample from t_b^1 to itself that satisfies the constant fairness constraint

and the constraint of β_j. By induction using the same arguments we can show that a counterexample for the extended concrete system exists. □

7.6. Simplifying the Correctness Property

In this section we describe how we can be transform the system $\mathcal{Q}(N)$ using the correctness property[6] $\Diamond\Box H \rightarrow \Diamond\Box J$. This step does not reduce the state space of the system. However, it reduces the number of reachable states and, more importantly, it allows us to express the correctness condition as a persistence property over the variables in $V_g \cup W_1$. Therefore, this step enables the abstraction of variables in $\bigcup_{i\in 2..N} V_{L(i)}$ in subsequent steps. We assume that $\mathcal{Q}(N) = (V, V, V, \Theta, \rho, L)$ is given by the parallel asynchronous composition of a set of processes, which can include a special process P_1, a generic process $P(i)$, and a process P_{init} that is used for the specification of additional initial conditions. Generic process $P(i) = (V_i, V_{L(i)}, V_{L(i)}, \Theta_i, \rho_i, L_i)$ is used for the instantiation of processes $P(2), ..., P(N)$. For the generic process we assume that the set of owned variables is equal to the set of local variables, which is true after the application of the first step. Moreover, because the set V_g of global variables of the system is fixed and finite, the size of the observable state space of each process does not depend on N.

The only condition for soundness and completeness of this step is that every reachable state that satisfies H is also an initial state. This is always the case for self-stabilizing systems, even after the application of the first step of our abstraction technique (Lemma 7.1).

[6]We will use H, $\mathcal{Q}(N)$, etc., instead of $\tilde{H}$, $\tilde{\mathcal{Q}}(N)$, etc., to denote the input to this step. We do that to simplify the description as the first step of the abstraction technique may not be necessary for some algorithms.

7.6.1. Transformation

Using the following lemma we can transform the first part of the correctness property to a safety property.

Lemma 7.2. *If every reachable state that satisfies H is an initial state, then for any N $\mathcal{Q}(N) \models \Diamond\Box H \rightarrow \Diamond\Box J$ if and only if $\mathcal{Q}(N) \models \Box H \rightarrow \Diamond\Box J$*

Proof. We start with the direction

$$(\mathcal{Q}(N) \models \Box H \rightarrow \Diamond\Box J) \Rightarrow (\mathcal{Q}(N) \models \Diamond\Box H \rightarrow \Diamond\Box J)$$

Suppose it holds $\mathcal{Q}(N) \models \Box H \rightarrow \Diamond\Box J$ and there is a behavior $\sigma = s_0, s_1, ...$ of $\mathcal{Q}(N)$ for which $\sigma \not\models \Diamond\Box H \rightarrow \Diamond\Box J$. Then

$$\sigma \not\models \neg(\Diamond\Box H) \vee \Diamond\Box J \quad \Rightarrow \quad \sigma \models \Diamond\Box H \wedge \Box\Diamond\neg J \quad \Rightarrow \quad \sigma \models \Diamond\Box H \ \wedge \ \sigma \models \Box\Diamond\neg J$$

Consequently, there exists $j \geq 0$ such that the execution segment starting at state s_j satisfies always H and has infinitely many $\neg J$ states. Since s_j is a reachable state satisfying H, it is also an initial state by the hypothesis. Therefore, there exists sequence $\tau = t_0, t_1, ...$ with $t_i = s_{j+i}, \forall i \geq 0$. Sequence τ is also a behavior of $\mathcal{Q}(N)$ and satisfies $\Box H \wedge \Box\Diamond\neg J$. However, that means that $\tau \not\models \Box H \rightarrow \Diamond\Box J$, which implies that $\mathcal{Q}(N) \not\models \Box H \rightarrow \Diamond\Box J$. This is a contradiction.

For the direction

$$(\mathcal{Q}(N) \models \Diamond\Box H \rightarrow \Diamond\Box J) \Rightarrow (\mathcal{Q}(N) \models \Box H \rightarrow \Diamond\Box J) \tag{7.2}$$

we note that any behavior σ of $\mathcal{Q}(N)$ that satisfies $\Box H$ satisfies $\Diamond\Box H$ as well. Therefore, for any σ such that $\sigma \models \Box H \wedge \Box\Diamond\neg J$ the following property holds $\sigma \models \Diamond\Box H \wedge \Box\Diamond\neg J$. Consequently, whenever the conclusion of the implication (7.2) is false, the hypothesis is false, too. $\Box$

From Lemma 7.2 we know that we can replace the correctness property with $\Box H \rightarrow \Diamond\Box J$. Any counterexample for the property $\Box H \rightarrow \Diamond\Box J$ must satisfy $\Box H$. Therefore, we can transform the system $\mathcal{Q}(N)$ to a new system $\breve{\mathcal{Q}}(N)$ whose computations are those computations of $\mathcal{Q}(N)$ that satisfy $\Box H$.

Lemma 7.3. *For the system $\breve{\mathcal{Q}}(N)$ which includes exactly those computations of $\mathcal{Q}(N)$ for which $\Box H$ holds, we have $\breve{\mathcal{Q}}(N) \models \Diamond\Box J \Leftrightarrow \mathcal{Q}(N) \models \Box H \rightarrow \Diamond\Box J$.*

System $\breve{\mathcal{Q}}(N)$ can be built by requiring that initially H holds and that any transition of the system preserves property H. More specifically, $\breve{\mathcal{Q}}(N)$ has initial condition $\breve{\Theta} = H \wedge \Theta$. In addition, from the relation ρ of $\mathcal{Q}(N)$, we can define the next state relation $\breve{\rho}$ of $\breve{\mathcal{Q}}(N)$ as $\breve{\rho} = \{\langle s, s'\rangle \in \rho \mid s' \models H\}$. The sets of variables and the liveness conditions of the two systems are the same. We can prove that every computation of $\mathcal{Q}(N)$ that is not a computation of $\breve{\mathcal{Q}}(N)$ has at least one state that violates H. Therefore, and from Lemmas 7.2 and 7.3, the theorem below follows.

Theorem 7.4. *If every reachable state that satisfies H is an initial state of $\mathcal{Q}(N)$, then for any N it holds $\mathcal{Q}(N) \models \Diamond\Box H \rightarrow \Diamond\Box J$ if and only if $\breve{\mathcal{Q}}(N) \models \Diamond\Box J$.*

One thing to notice about the new system $\breve{\mathcal{Q}}(N)$ is that all reachable states are initial states.

7.6.2. Action Based Transformation

The transformation described in the previous section is a sound and complete transformation for systems in which every reachable state is an initial state. This transformation can be used in some cases without additional modifications. However, there are cases in which we are required to preserve the locality constraint of the system. Moreover, we need the next state relation $\breve{\rho}$ of $\breve{\mathcal{Q}}(N)$ to be defined based on a set of actions for the subsequent steps of the methodology. In this section we define set of actions $\breve{A}(i)$ for each process $P(i)$ of $\breve{\mathcal{Q}}(N)$ from the set of actions $A(i)$ of $\mathcal{Q}(N)$ using the property H. In addition, we specify the conditions under which the locality constraint is preserved.

The transformation we describe in this section is a syntactic transformation of the actions. We change the next state expressions, so that an action is executed only if the next state satisfies H. Otherwise, a stuttering step is made. The weak and strong fairness conditions on the actions are the same as those in $\mathcal{Q}(N)$. The result is that the new system $\breve{\mathcal{Q}}(N)$ has exactly those computations of $\mathcal{Q}(N)$ that satisfy $\Box H$. The reason is that stuttering steps cannot satisfy the fairness constraints of the actions (Formula (7.1)). Therefore, a sequence σ of Σ^{ω} either satisfies or violates the liveness constraints of both systems.

Now we describe the action based transformation in detail. As we saw in Section 7.3 every action can be rewritten in disjunctive normal form. For each disjunct we create a predicate p from H by replacing the occurrence of each variable `var` with the expression that gives its value in the next state, if the action is executed. Then we replace the right-hand side $\epsilon(\alpha)$ of each proposition $\texttt{var}' = \epsilon(\alpha)$ by the expression $\text{ITE}(p, \epsilon(\alpha), \texttt{var})$, where ITE() stands for if-then-else operator. This operator returns the second argument,

if the first argument evaluates to true. Otherwise, it returns the third argument. If a proposition has the form $\texttt{var}' = \texttt{var}$, then the proposition remains unchanged.

If the next state satisfies the predicate H, then all conditions of the ITE expressions evaluate to true and the assignments happen as if the original action α was executed. However, if the next state does not satisfy H, then all conditions in the ITE expressions evaluate to false and the assignments of the original action cannot occur. In that case the variables are assigned the values they have in the current state ($\texttt{var}' = \texttt{var}$). The next state relation produced is a subset of the next state relation of the original system.

The syntactic transformation we described may violate the locality restriction. This is because the property check in the ITE expression may require the process to access variables local to other processes. We show that if H is the conjunction of a number of predicates, each expressed on a specific set of variables then locality is preserved. The reason is that each action includes in the ITE check only the predicates that it can violate. The following condition is sufficient for this purpose.

$\Phi 1$ H is of the form $H = \Gamma \wedge \Delta \wedge \forall i \in 2..N : E(i)$, where Γ is a predicate expressed over the variables in V_g, Δ is a predicate expressed over the variables in W_1, and $E(i)$ is a predicate expressed over the variables in $V_{L(i)}$.

This form is expected for systems that are composed of a number of identical up to renaming processes. Then $E(i)$ is the condition for each process, Δ is the condition for (the special) process 1, and Γ is the condition for the global variables. Note that if any of the predicates $\Gamma, \Delta, E(i)$ is missing, it can be replaced with True. Moreover, if H includes an additional conjunct that specifies the existence of a silent process, this conjunct can be removed after the first step of the abstraction technique. This type of conjunct is

normally of the form $\exists k \in 2..N : \texttt{lcr}[k] = v_k$ which is also an invariant for $P(k)$. If v_k is a constant value, a constant fairness constraint is added to the system during the first step, which guarantees fairness for the effect of reading v_k.

If $\Phi 1$ is satisfied, the checks of the ITE expressions happen only on the relevant variables. For example, an action of $P(i)$ can only violate Γ and $E(i)$ and, therefore, needs to check only those conditions to find if it preserves H. Therefore, the predicate check in the ITE expression can be expressed over the variables in V_g and $V_{L(i)}$ and the locality restriction is fulfilled. In addition, because the size of $V_g \cup V_{L(i)}$ is independent of N, the low-atomicity constraint is preserved. Consequently, the following theorem holds.

Theorem 7.5. *If property $\Phi 1$ holds, then the locality property of the distributed algorithm holds after the transformation.*

The algorithm for the syntactical transformation is displayed in Figures 7.3 and 7.4. The first part is a procedure for creating a disjunct of an action and the second part returns the new sets of actions $\breve{A}(i)$ and $\breve{A}_1$. The second set is the set of actions of the special process P_1 in the transformed system. The first set is the set of actions of the generic process based on which processes $P(2), ..., P(N)$ will be instantiated. In Figures 7.5 and 7.6 an example of the application of the transformation can be seen. In the next section we describe the approach of finding a network invariant for the transformed system.

```
proc create_disjunct(d,predicate)
p := predicate;
V_p := ∅;
foreach var appearing in p
   V_p := V_p ∪ {var};
   mark all occurrences of var in p;
endfor;
foreach var ∈ V_p
   let var' = ε(α) be a conjunct of d;
   replace all marked occurrences of
      var in p with ε(α);
endfor;
foreach conjunct c of d
   if c has the form var' = ε(α)
      if ε(α) is not var then
        replace c with var' =
          ITE(p, ε(α), var);
      fi;
   fi;
endfor;
Return d;
```

Figure 7.3. Procedure for creating a new disjunct using a predicate.

7.7. Finding a Network Invariant

In this section we present a method for building a network invariant for the system $\mathcal{Q}(N)$. For the system $\mathcal{Q}(N)$, which is the input to this step, we assume that every process has a finite number of observable variables and this number is independent of N. Moreover, we assume that the correctness property is expressed over the set W_1 of P_1's owned variables and that the size of this set is independent of N. Note that after the first two steps of the abstraction technique the system is expected to satisfy these properties. However, the number of processes in the system still depends on N and because of that the local state space of the processes, i.e., the variables in $V_L = \bigcup_{i \in 1..N} V_{L(i)}$, is different

Algorithm Create_set_$\breve{A}$_from_H
Input: The sets $A(i), A_1$ of actions of
the generic process $P(i)$ and
the special process P_1
Output: The sets of actions $\breve{A}(i)$ and $\breve{A}_1$
$\breve{A}(i) := \emptyset$;
for each action α_i of $A(i)$
$\breve{\alpha}_i :=$ False;
foreach disjunct d of α_i
$d_1 :=$ create_disjunct(d,$\Gamma \wedge E(i)$);
$\breve{\alpha}_i := \breve{\alpha}_i \vee d_1$;
endfor;
$\breve{A}(i) := \breve{A}(i) \cup \{\breve{\alpha}_i\}$;
endfor;
$\breve{A}_1 := \emptyset$;
for each action α of A_1
$\breve{\alpha} :=$ False;
foreach disjunct d of α
$d_1 :=$ create_disjunct(d,$\Gamma \wedge \Delta$);
$\breve{\alpha} := \breve{\alpha} \vee d_1$;
endfor;
$\breve{A}_1 := \breve{A}_1 \cup \{\breve{\alpha}\}$;
endfor;
Return $\breve{A}(i), \breve{A}_1$;

Figure 7.4. Algorithm for creating $\breve{A}(i), \breve{A}_1$.

for every instance of the system. In this section we reduce the local state space to a fixed finite set by building a network invariant.

We first define a few concepts which are important for the method of network invariants. We use the symbol $\sqsubseteq_M$ to denote modular abstraction [43]. Assume A and B are two processes. Then B is a modular abstraction of A, i.e., $A \sqsubseteq_M B$, if and only if A and B have the same set of observable variables and each observable behavior of A is an observable behavior of B for any environment. An observable behavior of A is a sequence obtained by projecting a behavior of the closed system $(A \| P_{env})$ on the set of observable

$H = \Gamma \wedge \Delta \wedge \bigwedge_{\forall k \in 2..N} : E(k)$
$\Gamma \triangleq (\texttt{gdis} = min\_dis) \Leftrightarrow (\texttt{gid} = fid)$
$E(k) \triangleq (\texttt{ldis}[k] = min\_dis) \Leftrightarrow (\texttt{lid}[k] = fid)$
$\Delta \triangleq E(1)$
$\texttt{ldis}[k], \texttt{cdis}[k], \texttt{gdis} \in \{min\_dis, gt\_min\_dis, any\_dis\}$
$\texttt{lid}[k], \texttt{cid}[k], \texttt{gid} \in \{fid, oid\}$

Initial condition: $\Theta = \texttt{True}$
Actions of process $P(i)$ **for all** $i \in 1..N$

$\alpha \triangleq \wedge \texttt{cid}[i]' = \texttt{gid}$
$\wedge \texttt{cdis}[i]' = \texttt{gdis}$

$\beta \triangleq \wedge \texttt{cdis}[i] = min\_dis$
$\wedge \texttt{gid}' = \texttt{cid}[i]$
$\wedge \texttt{lid}[i]' = \texttt{cid}[i]$
$\wedge \texttt{gdis}' = gt\_min\_dis$
$\wedge \texttt{ldis}[i]' = gt\_min\_dis$

$\gamma \triangleq \wedge \texttt{cdis}[i] = any\_dis$
$\wedge \texttt{gid}' = \texttt{cid}[i]$
$\wedge \texttt{lid}[i]' = \texttt{cid}[i]$
$\wedge \texttt{gdis}' = any\_dis$
$\wedge \texttt{ldis}[i]' = any\_dis$

$\delta \triangleq \wedge \texttt{gid}' = \texttt{lid}[i]$
$\wedge \texttt{gdis}' = \texttt{ldis}[i]$

Figure 7.5. The initial system and the processes before the second step of the abstraction technique. For the variables we have $V_g = \{\texttt{gdis}, \texttt{gid}\}$ and $V_{L(i)} = \{\texttt{cid[i]}, \texttt{cdis[i]}, \texttt{lid[i]}, \texttt{ldis[i]}\}$. The fairness constraints and the part of the actions that describe which variables are left unchanged are not shown.

variables of A, where P_{env} is the specification of the environment. Formally, if A and B are defined as $A = (V_A, W_A, V_{LA}, \Theta_A, \rho_A, L_A)$, $B = (V_B, W_B, V_{LB}, \Theta_B, \rho_B, L_B)$, we have

$$A \sqsubseteq_M B \Leftrightarrow \begin{cases} \wedge \quad V_A - V_{LA} = V_B - V_{LB} & \text{(Same set of observable variables)} \\ \wedge \quad \forall P_{env} : \forall \sigma : \sigma \models (A \| P_{env}) \Rightarrow \exists \tau : \quad \wedge \quad \Pi_{(V_A - V_{LA})}(\sigma) = \Pi_{(V_B - V_{LB})}(\tau) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad \tau \models (B \| P_{env}) \end{cases} \tag{7.3}$$

Initial condition: $\Theta = H$
Actions of process $P(i)$ **for all** $i \in 1..N$

$$
\begin{array}{lcl}
\alpha & \triangleq & \wedge \texttt{cid}[i]' = \texttt{gid} \\
 & & \wedge \texttt{cdis}[i]' = \texttt{gdis} \\
\beta & \triangleq & \wedge \texttt{cdis}[i] = \mathit{min\_dis} \\
 & & \wedge \texttt{gid}' = \texttt{ITE}(\texttt{cid}[i] = \mathit{fid}, \texttt{cid}[i], \texttt{gid}) \\
 & & \wedge \texttt{lid}[i]' = \texttt{ITE}(\texttt{cid}[i] = \mathit{fid}, \texttt{cid}[i], \texttt{lid}[i]) \\
 & & \wedge \texttt{gdis}' = \texttt{ITE}(\texttt{cid}[i] = \mathit{fid}, \mathit{gt\_min\_dis}, \texttt{gdis}) \\
 & & \wedge \texttt{ldis}[i]' = \texttt{ITE}(\texttt{cid}[i] = \mathit{fid}, \mathit{gt\_min\_dis}, \texttt{ldis}[i]) \\
\gamma & \triangleq & \wedge \texttt{cdis}[i] = \mathit{any\_dis} \\
 & & \wedge \texttt{gid}' = \texttt{ITE}(\texttt{cid}[i] \neq \mathit{fid}, \texttt{cid}[i], \texttt{gid}) \\
 & & \wedge \texttt{lid}[i]' = \texttt{ITE}(\texttt{cid}[i] \neq \mathit{fid}, \texttt{cid}[i], \texttt{lid}[i]) \\
 & & \wedge \texttt{gdis}' = \texttt{ITE}(\texttt{cid}[i] \neq \mathit{fid}, \mathit{gt\_min\_dis}, \texttt{gdis}) \\
 & & \wedge \texttt{ldis}[i]' = \texttt{ITE}(\texttt{cid}[i] \neq \mathit{fid}, \mathit{gt\_min\_dis}, \texttt{ldis}[i]) \\
\delta & \triangleq & \wedge \texttt{gid}' = \texttt{ITE}(E(i), \texttt{lid}[i], \texttt{gid}) \\
 & & \wedge \texttt{gdis}' = \texttt{ITE}(E(i), \texttt{ldis}[i], \texttt{gdis})
\end{array}
$$

Figure 7.6. The system of Figure 7.5 after the application of the second step of the abstraction technique.

In (7.3) P_{env} is any process composable with both A and B, i.e., any process that does not change the variables in W_A and W_B. This process describes the behavior of the environment. Sequence σ is a behavior of the system $(A \| P_{env})$ and τ is a behavior of the system $(B \| P_{env})$. An important property of modular abstraction is that for any process C which is composable with both A and B we have

$$A \sqsubseteq_M B \;\Rightarrow\; A \| C \sqsubseteq_M B \| C$$

A network invariant I is a special process that is used as a modular abstraction of an arbitrary number of symmetric processes [45, 43, 52]. We use the network invariant to abstract the processes $P(2), ..., P(N)$ which are all created from the specification of the generic process $P(i)$. In order for a process I to be a correct network invariant it must

satisfy the following two conditions:

$$P(i) \quad \sqsubseteq_M \quad I \qquad (7.4) \qquad\qquad I||I \quad \sqsubseteq_M \quad I \quad (7.5)$$

In this section we show how the user can exploit property H to create a process I_H that fulfills (7.4) and (7.5). Property H is important because it restricts the possible behaviors of the environment. Therefore, even though the conditions (7.4) and (7.5) have to hold for any environment, we know that the specific environment of each group of processes $P(2), ..., P(N)$ is part of the system $\mathcal{Q}(N)$ that always satisfies H. Because of that, process I_H has two operational modes; normal and chaotic. Process I_H is in the normal mode, as long as it observes its environment satisfying H. In this mode the observable behavior of I_H is consistent with the actions of the generic process $P(i)$. Moreover, I_H preserves H when assigning new values to its local variables. Process I_H enters the chaotic mode only if the environment violates H. In that mode I_H's actions are restricted only by ownership conditions.

We formally define I_H as $I_H \triangleq (V_I, V_{LI}, V_{LI}, \Theta_I, \rho_I, \text{True})$ and describe how it can be obtained from the specification of the generic process $P(k) = (V_k, V_{L(k)}, V_{L(k)}, \Theta_k, \rho_k, L_k)$:

V_{LI}: The set V_{LI} of local variables is obtained by the set $V_{L(k)}$ of the generic process $P(k)$ by renaming each variable $\texttt{var}[k]$ as $\texttt{var}_I$. Moreover, a new variable `mode` is added to that set

$$V_{LI} \triangleq \{\texttt{var}_I | \texttt{var}[k] \in V_{L(k)}\} \cup \{\texttt{mode}\}$$

V_I: The set V_I of variables I_H can read or modify is given by

$$V_I \triangleq (V_k - V_{L(k)}) \cup V_{LI} \tag{7.6}$$

The set $V_k - V_{L(k)}$ contains the variables $\texttt{cr}_a[1]$ and $\texttt{cr}_a[2]$ after the first step of our abstraction technique.

Θ_I: The initial condition Θ_I is the conjunct of two conditions. The first is obtained from Θ_k by replacing the occurrence of each variable $\texttt{var}[k] \in V_{L(k)}$ with the corresponding variables $\texttt{var}_I \in V_{LI}$. The second is the predicate $\texttt{mode} = \text{normal}$, which specifies that initially I_H is in the normal operation mode.

ρ_I: The next state relation ρ_I is built from a new set of actions A_I. Process I_H starts in the normal mode, in which its actions have the same observable behavior as the actions of the generic process $P(k)$. It changes to chaotic mode, once it observes that the environment has violated property H. The actions of the normal operation mode are built from the actions of $A(k)$. The actions in the chaotic mode are less restricted.

We start with the actions in the normal mode. For each action $\alpha \in A(k)$ we create a new action α_i by generating one by one its disjuncts from the disjuncts of α. More specifically, for each disjunct of α the conjuncts that describe the next state values of local variables are replaced with conjuncts that assign any value in the domain of the variables. For example, a conjunct of the form $\texttt{var}[k]' = \epsilon(\alpha)$ is replaced with $\texttt{var}'_I \in D_{\texttt{var}[k]}$, where $D_{\texttt{var}[k]}$ is the domain of $\texttt{var}[k]$ and $\texttt{var}_I$ is the renamed version of $\texttt{var}[k]$. In all other conjuncts the only change is that the local variables are replaced with their renamed versions (Figures 7.7

and 7.8). Then the action is brought back into disjunctive normal form. Since we do not want the next local state to be a state that violates H, we preserve H by the method we described in Section 7.6.2. More specifically, we replace the assignments to the local variables by ITE, i.e., if-then-else, expressions, whose checks guarantee that if H could be violated by the disjunct, the action executed is the stuttering step. If H satisfies $\Phi 1$ (Section 7.6), then the conjuncts that are used in the ITE check are Γ, which is the conjunct defined over the global variables[7], and E, which is the conjunct specified over the local variables V_{LI}. Predicate E is obtained from $E(k)$ by variable renaming.

The actions obtained by the procedure above have observable behavior that is consistent with the actions of $P(k)$. The reason is that on the conjuncts that specify the next state values of global variables only renaming was done. Therefore, the effect of each action on those variables is consistent with the effect the action has when executed by a generic process that is in the same the local state as I_H. The next local state of I_H is any state that satisfies H.

To the precondition of each of these actions conjuncts ($\texttt{mode}$ = normal) and Γ are added. This is because these actions are executed only in the normal mode and only when the Γ conjunct of H is satisfied. Process I_H cannot check the Δ conjunct of H, as it is expressed over the local variables of $P(1)$. Moreover, E is expressed over its own local variables and the environment cannot violate it. Conjunct $\texttt{mode}' = \texttt{mode}$ is added to all disjuncts of all actions of the normal mode.

[7] In Γ two conjuncts for the valid values of $\texttt{cr}_a[1]$ and $\texttt{cr}_a[2]$ are added.

An action α_{n2c} is included in A_I for the transition from the normal mode to the chaotic mode. This action is enabled when after the environment of I_H violates Γ in the normal mode, i.e., $\texttt{mode} = \text{normal} \ \wedge \ \neg\Gamma$, and assigns to all variables that I_H can modify any value in their domains. More specifically, a conjunct $\texttt{var}' \in D_{\texttt{var}}$ is added for each variable $\texttt{var} \in V_I - \{\texttt{cr}_a[1], \texttt{mode}\}$. The value chaotic is assigned to variable mode, as I_H enters the chaotic mode.

A single action α_c is included in A_I for the chaotic mode. It is enabled when $\texttt{mode} = \text{chaotic}$ and it assigns to each variable $\texttt{var} \in V_I - \{\texttt{cr}_a[1], \texttt{mode}\}$ any value in their domains. Variable mode remains unchanged, as I_H cannot return to the normal mode. The algorithm for the creation of A_I from $A(k)$ is displayed in Figure 7.9.

There is no liveness condition for process I_H. The following theorem states that I_H is a network invariant of any group of processes created from the generic process $P(k)$.

Theorem 7.6. *The system I_H is a network invariant for the processes $P(2)\|...\|P(N)$ for any $N \in \mathbb{N}$.*

Proof. We need to show that I_H satisfies (7.4) and (7.5). We start with (7.4). Processes $P(k)$ and I_H have the same set of observable variables by construction. Now assume that P_{env} is any process that is composable with $P(k)$ and I_H. We denote as Θ_{env} its initial condition. We show that for any behavior $\sigma = s_0, s_1, ...$ of $(P(k)\|P_{env})$ there exists a behavior $\tau = t_0, t_1, ...$ of $(I_H\|P_{env})$ such that $\forall n \geq 0 : \Pi_{V_k - V_{L(k)}}(s_n) = \Pi_{V_I - V_{LI}}(t_n)$.

We first notice that for every variable of $(P(k)\|P_{env})$ there is a corresponding variable in $(I_H\|P_{env})$. System $(I_H\|P_{env})$ has the additional variable mode to which none of the

```
proc transform_conjunct(c)
foreach var[k] ∈ V_{L(k)}
   replace each occurrence of var[k] in c
   with the corresponding variable of V_LI;
endfor;
Return c;
```

Figure 7.7. Procedure for transforming a conjunct c.

```
proc transform_disjunct(d_k)
d := True;
foreach conjunct c of d_k
   if ∃var[k] ∈ V_{L(k)} that appears in c
      if c is part of eff(α_k)
        let c = (var' := ε(α_k));
        if var ∉ V_{L(k)}
          c :=transform_conjunct(c);
          d := d ∧ c;
        endif;
      elseif c is part of prec(α)
        c :=transform_conjunct(c);
        d := d ∧ c;
      endif;
   else
      d := d ∧ c;
   endif;
enfor;
foreach var_I in V_LI
   let D_{var_I} be the domain of var_I;
   create conjunct c_new := (var'_I ∈ D_{var_I});
   d := d ∧ c_new;
endfor;
Return d;
```

Figure 7.8. Procedure for transforming a disjunct d.

variables of the former system is mapped to. We know that $s_0 \models \Theta_k \wedge \Theta_{env}$, so we build t_0 with the same values for all corresponding variables and `mode` = normal. For every transition (s_i, s_{i+1}) we build a transition (t_i, t_{i+1}) with $\Pi_{V_k - V_{L(k)}}(s_{i+1}) = \Pi_{V_I - V_{LI}}(t_{i+1})$ as follows:

Algorithm `Create_set_`A_I`_using_H`
Input: `The set` $A(k)$ `of actions of process` $P(k)$
Output: `The set of actions` A_I
$A_I := \emptyset$;
`for each action` α_k `of` $A(k)$
  $\alpha :=$ `False`;
  `foreach disjunct` d_k `of` α_k
    $d :=$ `transform_disjunct(`d_k`)`;
    $\alpha := \alpha \vee d$;
  `endfor`;
  `bring` α `into dfn`;
  $A_I := A_I \cup \{\alpha\}$;
`endfor`;
`foreach` α `in` A_I
  $A_I := A_I - \{\alpha\}$;
  `foreach disjunct` d `of` α
    $d := d \wedge (\texttt{mode} = \texttt{normal}) \wedge \Gamma$;
    $d :=$ `create_disjunct(`d`,`$\Gamma \wedge E$`)`;
  `endfor`;
  $A_I := A_I \cup \{\alpha\}$;
`endfor`;
$\alpha_{n2c} :=$ `False`;
`foreach` $var \in V_I \cup V_{LI}$
  `let` D_{var} `be the domain of` var;
  $c := var' \in D_{var}$;
  $\alpha_{n2c} := \alpha_{n2c} \wedge c$;
`endfor`;
$\alpha_{n2c} := \alpha_{n2c} \wedge (\texttt{mode} = \texttt{normal}) \wedge \neg\Gamma \wedge (\texttt{mode}' = \texttt{chaotic})$;
$A_I := A_I \cup \{\alpha_{n2c}\}$;
$\alpha_c :=$ `False`;
`foreach` $var \in V_I \cup V_{LI}$
  `let` D_{var} `be the domain of` var;
  $c := var' \in D_{var}$;
  $\alpha_c := \alpha_c \wedge c$;
`endfor`;
$\alpha_c := \alpha_c \wedge (\texttt{mode} = \texttt{chaotic})$;
$A_I := A_I \cup \{\alpha_c\}$;
`Return` A_I;

Figure 7.9. Algorithm for creating A_I. The function `create_disjunct` is displayed in Figure 7.3.

$D_{id} \triangleq \{\texttt{fid}, \texttt{oid}\}$
$D_{dis} \triangleq \{\texttt{min\_dis}, \texttt{any\_dis}, \texttt{gt\_min\_dis}\}$
Actions of I_H

$\alpha \triangleq \texttt{mode} = \texttt{normal}$
$\quad \wedge \Gamma$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{fid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' = \texttt{min\_dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{oid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in \{\texttt{any\_dis}, \texttt{gt\_min\_dis}\}$

$\beta \triangleq \wedge \texttt{mode} = \texttt{normal}$
$\quad \wedge \Gamma$
$\quad \texttt{cdis}_{\texttt{I}_\texttt{H}} = \texttt{min\_dis}$
$\quad \wedge \texttt{gid}' = \texttt{ITE}(\texttt{cid}_{\texttt{I}_\texttt{H}} = \texttt{fid}, \texttt{cid}_{\texttt{I}_\texttt{H}}, \texttt{gid})$
$\quad \wedge \texttt{gdis}' = \texttt{ITE}(\texttt{cid}_{\texttt{I}_\texttt{H}} = \texttt{fid}, \texttt{gt\_min\_dis}, \texttt{gdis})$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{fid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' = \texttt{min\_dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{oid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in \{\texttt{any\_dis}, \texttt{gt\_min\_dis}\}$

$\gamma \triangleq \wedge \texttt{mode} = \texttt{normal}$
$\quad \wedge \Gamma$
$\quad \texttt{cdis}_{\texttt{I}_\texttt{H}} = \texttt{any\_dis}$
$\quad \wedge \texttt{gid}' = \texttt{ITE}(\texttt{cid}_{\texttt{I}_\texttt{H}} \neq \texttt{fid}, \texttt{cid}_{\texttt{I}_\texttt{H}}, \texttt{gid})$
$\quad \wedge \texttt{gdis}' = \texttt{ITE}(\texttt{cid}_{\texttt{I}_\texttt{H}} \neq \texttt{fid}, \texttt{gt\_min\_dis}, \texttt{gdis})$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{fid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' = \texttt{min\_dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{oid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in \{\texttt{any\_dis}, \texttt{gt\_min\_dis}\}$

$\delta \triangleq \wedge \texttt{mode} = \texttt{normal}$
$\quad \wedge \Gamma$
$\quad \wedge \texttt{gid}' = \texttt{ITE}(E(i), \texttt{lid}_{\texttt{I}_\texttt{H}}, \texttt{gid})$
$\quad \wedge \texttt{gdis}' = \texttt{ITE}(E(i), \texttt{ldis}_{\texttt{I}_\texttt{H}}, \texttt{gdis})$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{fid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' = \texttt{min\_dis}$
$\quad \wedge \vee \texttt{lid}_{\texttt{I}_\texttt{H}}{}' = \texttt{oid}$
$\quad\quad \vee \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in$
$\quad\quad\quad \{\texttt{any\_dis}, \texttt{gt\_min\_dis}\}$

$\alpha_{n2c} \triangleq \wedge \texttt{mode} = \texttt{normal}$
$\quad \wedge \neg\Gamma$
$\quad \wedge \texttt{gid}' \in D_{id}$
$\quad \wedge \texttt{gdis}' \in D_{dis}$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \texttt{lid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \texttt{mode}' = \texttt{chaotic}$

$\alpha_2 \triangleq \wedge \texttt{mode} = \texttt{chaotic}$
$\quad \wedge \texttt{gid}' \in D_{id}$
$\quad \wedge \texttt{gdis}' \in D_{dis}$
$\quad \wedge \texttt{cid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{cdis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$
$\quad \wedge \texttt{lid}_{\texttt{I}_\texttt{H}}{}' \in D_{id}$
$\quad \wedge \texttt{ldis}_{\texttt{I}_\texttt{H}}{}' \in D_{dis}$

Figure 7.10. The actions of the network invariant I_H for the example of Figure 7.6. For simplicity the actions are shown in a more compact form than disjunctive normal form and the part of the actions describing the variables left unchanged is omitted.

(1) If P_{env} executes the action that causes the transition, the same action is enabled in t_i and its execution causes $(I_H \| P_{env})$ to move to a state t_{i+1} in which all corresponding variables have the same values.

(2) Let $P(k)$ be the process that executes the action α that causes the transition. If Γ has not been violated before any action of $P(k)$ executed so far, then α preserves H. For any such action, there is a corresponding action α_I of the normal mode of I_H with one of its disjuncts having the same effect as α. State t_{i+1} is obtained by choosing this disjunct of α_I. Action α_I is enabled, as Γ has not been violated before actions of I_H and, therefore, I_H is in the normal mode.

(3) Let $P(k)$ be the process that executes the action α that causes the transition. If Γ has been violated before an action of $P(k)$, then we cannot guarantee that α preserves H. However, in that case I_H is or enters the chaotic mode. For every effect that α has, the actions α_{n2c} and α_c have a disjunct that produces the same effect on the corresponding variables. Therefore, there exists t_{i+1} as a result of those actions.

Since I_H has no liveness condition, it is a modular abstraction of the generic process $P(i)$.

The next step is to show that $I_H \| I_H \sqsubseteq_M I_H$. First, we notice that I_H does not have any observable owned variables. Therefore, by renaming the local variables of two processes created from the specification of I_H, these two processes become composable. More specifically, we use ids from the set $\{1, 2\}$ for naming convenience and rename each $\mathtt{var}_I \in V_{LI}$ as $\mathtt{var}_I[1]$ for one process and as $\mathtt{var}_I[2]$ for the other. Then we augment the system $I_H \| I_H$ with prophecy variable $\pi \in \{1, 2\}$. This variable stores the id of the next of the two processes to execute a step. Every action of I_H with id$= 1$ is guarded by

$\pi = 1$ and every action of the other process is guarded by $\pi = 2$. After every action π can take any value in $\{1, 2\}$. The π variable does not affect the behavior of the two processes and is only used as a prophecy variable. Moreover, the assigned ids are used only for naming convenience and do not affect the observable behavior of the system $I_H \| I_H$. To distinguish between the two processes we denote them as $I_H[1]$ and $I_H[2]$.

In the same way as in the first part of the proof, we show that $I_H[1] \| I_H[2] \sqsubseteq_M I_H$. More specifically, we show that for any process P_{env} that is composable with I_H and for any behavior $\sigma = s_0, s_1, ...$ of the system $(I_H[1] \| I_H[2] \| P_{env})$, there is a behavior $\tau = t_0, t_1, ...$ of $(I_H \| P_{env})$ such that $\forall n \geq 0 : \Pi_{V_I - V_{LI[1]} - V_{LI[2]}}(s_n) = \Pi_{V_I - V_{LI}}(t_n)$. We call $(I_H[1] \| I_H[2] \| P_{env})$ the first system and $(I_H \| P_{env})$ the second.

There is an one-to-one mapping between the observable variables of the two systems. The same holds for all local variables of P_{env}. For the variables that are local to the two I_H processes of the first system and the I_H of the second, we create the following relation $\forall n \geq 0 : \Pi_{\mathtt{var}_{I[\pi]}}(s_i) = \Pi_{\mathtt{var}_I}(t_i)$ for every variable $\mathtt{var}_I \in V_{LI} - \{\mathtt{mode}\}$. That means that in the second system all local variables of I_H, except `mode`, have the same values as the local variables of the I_H process that will execute the next step in the first system, i.e., $I_H[\pi]$.

State s_0 satisfies $\Theta_{env} \wedge \Theta_{I[1]} \wedge \Theta_{I[2]} \wedge \pi \in \{1, 2\}$. We build state t_0 by assigning to each corresponding variable the same value. Moreover, for the local variables of I_H we assign values equal to the values of the local variables of $I_H[\pi]$. For every transition (s_i, s_{i+1}) we build a transition (t_i, t_{i+1}) with $\Pi_{V_I - V_{LI[1]} - V_{LI[2]}}(s_{i+1}) = \Pi_{V_I - V_{LI}}(t_{i+1})$ and $\Pi_{V_{LI[\pi]} - \{\mathtt{mode}\}}(s_i) = \Pi_{V_{LI} - \{\mathtt{mode}\}}(t_i)$ as follows:

(1) If P_{env} executes the action that causes the transition, the same action is enabled in t_i and its execution causes $(I_H \| P_{env})$ to move to a state t_{i+1} in which all corresponding variables have the same values. The local variables of I_H and $I_H[\pi]$ do not change, so their relation is preserved.

(2) Assume that $I_H[\pi]$ executes an action α that causes the transition and $\texttt{mode}[1] = \texttt{mode}[2] =$ normal in s_{i+1}. Then I_H is also in the normal mode, as before the execution of its actions in this behavior Γ is satisfied. Moreover, the global variables have the same values and the values of the variables in $V_{I[\pi]}$ are the same as the values of the variables in V_I. Hence, process I_H can execute one of the actions with the same effect on the global variables. If α sets π' to 1, the next local state of I_H is the same as that of $I_H[1]$, else it is the same as the $I_H[2]$. This is possible because both processes have a local state that satisfies H, so there is some disjunct of the action to bring I_H to either state.

(3) Assume that $I_H[\pi]$ executes an action α that causes the transition and $\texttt{mode}[1] =$ chaotic$\vee\texttt{mode}[2] =$ chaotic in s_{i+1}. Then I_H is also in the chaotic mode in t_{i+1}, as before one of its actions, Γ has been violated. Since either α_{n2c} or α_c is executed in t_i, process I_H has less restrictions. The effect of the action it executes can be the same as $I_H[\pi]$. Moreover, the next local state is the local state of $I_H[\pi']$ independent of whether this state violates H. The variable mode of I_H remains to chaotic for the rest of the behavior.

Since there is no liveness condition, I_H is a modular abstraction of $I_H[1] \| I_H[2]$. $\square$

Local variables that are used neither in the precondition of any action nor in the computation of the next state value of a variable can be eliminated for further reduction of the state space of I_H.

7.7.1. Completeness

In this section we present sufficient conditions for completeness of the third step. The following conditions are sufficient for the construction of a counterexample for system $\mathcal{Q}(N)$ from a counterexample of the system $(P(1)\|I_H)$:

- C1 Every reachable state of the system is an initial state.
- C2 The number of processes in the system is not bounded from above.
- C3 For all N_1 and N_2 with $N_1 < N_2$, the systems $\mathcal{Q}(N_1)$ and $\mathcal{Q}(N_2)$ have the same formula as liveness constraint.
- C4 Actions of $P(k)$ for $k \in 2..N$ that modify variables in V_g do not modify variables in $V_{L(k)}$.

Condition C1 is always true for system $\mathcal{Q}(N)$. Conditions C2 and C3 have been discussed in Section 7.5.2. Condition C4 is satisfied by systems in which any write action on one of the variables in V_g preserves the local state space of the process.

Theorem 7.7. *If $\mathcal{Q}(N)$ satisfies conditions C1 – C4 and $(P(1)\|I_H) \not\models \Diamond\Box J$, then $\exists K \in \mathbb{N}$ such that $\mathcal{Q}(K) \not\models \Diamond\Box J$.*

Proof. We assume we have a counterexample for the system $(P(1)\|I_H)$. As in the proof of Theorem 7.1 we focus only on the cycle of the counterexample (C1). For every action that I_H executes in this cycle, we add a process $P(k)$ in the system $\mathcal{Q}(N)$ with

the same local state as the local state of I_H before the action. Because of C1 such a state is a valid initial state for process $P(k)$. If C4 is satisfied, then the addition of the new process does not prevent us from forming a cycle. This is because each process $P(k)$ has the same local state before and after the action and does not need to execute any other action to maintain this local state. Because of C3 no additional fairness constraints are added to the extended system. Moreover, condition C2 allows the addition of processes for any number of actions I_H executes in the cycle. For each action of $P(1)$ in the abstract counterexample, there is a corresponding action in the concrete system. The cycle formed this way is a behavior of the system[8] $\mathcal{Q}(K)$ for $K \in \mathbb{N}$. Since the property J is specified over the variables in W_1, the cycle formed in the state transition graph of $\mathcal{Q}(K)$ violates $\Diamond\Box J$. $\Box$

7.8. Case Studies

In this section we demonstrate the effectiveness of our technique on 3 self-stabilizing algorithms; spanning-tree construction, leader-election, and coloring. All 3 case studies satisfy the conditions for soundness after the preprocessing step. The conditions for completeness are not necessary, but sufficient for the automated creation of the concrete counterexample. The algorithms presented here do not satisfy all conditions for completeness. Therefore, we cannot exclude the possibility of a spurious counterexample. However, proving correctness in all 3 cases demonstrates the effectiveness of our approach as a sound abstraction technique.

[8]As mentioned above each process of system $\mathcal{Q}(K)$ has a fixed number of observable variables because of the transformations applied during the previous steps of the technique.

7.8.1. Spanning-Tree Construction

The first example we applied our abstraction technique is a variant of Arora and Gouda's low-atomicity spanning-tree algorithm [3]. In this algorithm each node $P(i)$ stores the root of the tree in variable $\texttt{root}[i]$, its distance from the root in $\texttt{dis}[i]$, and its parent in the tree in $\texttt{F}[i]$. The parent is the neighbor node with the minimum distance from the root. The root of the tree becomes the node with the maximum id in the graph. The communication register of $P(i)$ is defined as $\texttt{cr}[i] \triangleq (\texttt{root}[i], \texttt{dis}[i])$.

In the original low-atomicity spanning tree algorithm, each process stores local copies of the communication registers of all its neighbors. Clearly, in this case the number of local variables of process $P(i)$ depends on the number of its neighbors. Therefore, our technique cannot be applied on the original version of the algorithm. We replace this set of variables with only one local copy $\texttt{ncre}[i]$. Each process i is given one action per neighbor j that copies the communication register of the neighbor to the local copy, i.e., $\texttt{ncre}[i]' = \texttt{cr}[j]$. All actions of the process have strong fairness conditions.

Another problem with the original algorithm is that the variables have large or infinite domains. We abstract the system, so that the variables take values from finite sets. More specifically, the domain of the root variables becomes $D_{\texttt{root}} = \{$ `LtR` , `EqR` , `GtR` $\}$, where `LtR` , `EqR` , and `GtR` stand for "less than the root id", "equal to the root id", and "greater than the root id", respectively. For the distance, we assume that the special process $P(1)$ is at distance l from the root and we reduce the set to a few values of interest, i.e., $D_{\texttt{dis}} = \{\bot, l-2, l-1, l, l+1, \top\}$. Finally, the domain of the parent field is abstracted to $D_{\texttt{F}} = \{$ `node_1` , `not_neighbor` , `neighbor_l-1` , `neighbor_geq_l` $\}$.

The subgraph obtained by process $P(1)$ and its neighbors is not a closed system, but part of an arbitrary graph. We make the system closed by adding an action to each of the processes $P(2), ..., P(N)$ that has the same effect as the action reading any valid value of the removed processes. Finally, because communication registers are used in the preconditions of some actions, we add local copies of the communication registers for all processes.

The system obtained after the preprocessing step is amenable to our technique. It satisfies all constraints for soundness and is a silent self-stabilizing algorithm. We want to prove that after all nodes in distance $l-1$ from the root have stabilized, then eventually a node in distance l from the root will stabilize. More specifically, property H specifies that floating root ids have been eliminated, the nodes that are in distance $l-1$ from the root have stabilized, and all other nodes that have identified the root hold distance values greater or equal to l. Property J specifies that the node $P(1)$ in distance l stabilizes. Due to symmetry we can conclude that all nodes in distance l stabilize. Formally, we want to

prove that for this system $\Diamond\Box H \rightarrow \Diamond\Box J$, where

$$
\begin{aligned}
H \triangleq \quad & \wedge \quad \texttt{lcr}[1] = \texttt{cr}[1] \\
& \wedge \quad \texttt{lcr}[1].\texttt{root} \in \{\ \texttt{LtR}\ ,\ \texttt{EqR}\ \} \\
& \wedge \quad \texttt{lcr}[1].\texttt{root} = \ \texttt{EqR}\ \Rightarrow \texttt{lcr}[1].\texttt{dis} \geq l \\
& \wedge \quad \texttt{ncre}[1].\texttt{root} \in \{\ \texttt{LtR}\ ,\ \texttt{EqR}\ \} \\
& \bigwedge_{j \in 2..N} \quad \texttt{lcr}[j].\texttt{root} \in \{\ \texttt{LtR}\ ,\ \texttt{EqR}\ \} \\
& \bigwedge_{j \in 2..N} \quad \texttt{lcr}[j].\texttt{root} = \ \texttt{EqR}\ \Rightarrow \texttt{lcr}[j].\texttt{dis} \geq l - 1 \\
& \bigwedge_{j \in 2..N} \quad \texttt{ncre}[j].\texttt{root} \in \{\ \texttt{LtR}\ ,\ \texttt{EqR}\ \} \\
& \bigwedge_{j \in 2..N} \quad \texttt{nF}[j] \neq \ \texttt{not\_neighbor}\ \Rightarrow (\texttt{ncre}[j].\texttt{root} = \ \texttt{EqR}\ \Rightarrow \texttt{lcr}[j].\texttt{dis} \geq l - 1) \\
& \bigwedge_{j \in 2..N} \quad \texttt{nF}[j] = \ \texttt{not\_neighbor}\ \Rightarrow (\texttt{ncre}[j].\texttt{root} = \ \texttt{EqR}\ \Rightarrow \texttt{lcr}[j].\texttt{dis} \geq l - 2) \\
& \wedge \quad \exists j \in 2..N : \texttt{lcr}[j].\texttt{root} = \ \texttt{EqR}\ \wedge \texttt{lcr}[j].\texttt{dis} = l - 1 \wedge \texttt{F}[j] \in \{\ \texttt{not\_neighbor}\ \}
\end{aligned}
$$

$$
J \triangleq \ \texttt{lcr}[1].\texttt{root} = \ \texttt{EqR}\ \wedge \texttt{lcr}[1].\texttt{dis} = l \wedge \texttt{F}[1] \in \{\ \texttt{neighbor\_l-1}\ \ \}
$$

The first 4 conjuncts of H form Δ, which is the condition specified for the special process $P(1)$. The first one specifies that the communication register and its local copy have the same value, or equivalently, process $P(1)$ has executed its first write action since the beginning of the execution. The second specifies that any floating id that is greater than the maximum id in the graph, which is the id of the root, has been eliminated. The third is an inductive property, which specifies that if $P(1)$ has the correct id value then

its distance value is greater or equal to l. The fourth conjuncts specifies that the variable `ncre`[1], which holds the value of one of its neighbors, does not contain a floating id.

The next 5 conjuncts of H form $\forall k \in 2..N : E(k)$. The first three are similar to conjuncts contained in Δ. The next 2 determine whether the last value read by each of those processes belongs to a process outside of the closed system (`not_neighbor`), where processes of distance $l-2$ can be found. The last conjunct of H specifies that there is at least one neighbor of $P(1)$ that is at distance $l-1$ from the root and has stabilized. Property J specifies that $P(1)$ stabilizes.

During the first step of our technique a constant fairness condition is added. Since this a silent self-stabilizing system, the set of $N-1$ read actions of process $P(1)$, i.e.,

$\exists j \in 2..N \quad :$

$$\begin{aligned} \alpha_r(j) \quad \triangleq \quad & \wedge \quad \texttt{ncre}[1]' = \texttt{cr}[j] \\ & \wedge \quad \texttt{nF}[1]' = \text{ITE}(\texttt{cr}[j] = (\ \texttt{EqR}\ , l-1),\ \texttt{neighbor\_l-1}\ \ ,\ \texttt{neighbor\_geq\_l}\ ) \\ & \bigwedge_{\texttt{var} \in V - \{\texttt{ncre}[1], \texttt{nF}[1]\}} \texttt{var}' = \texttt{var} \end{aligned}$$

create the constant fairness constraint $cf(\alpha_r) \triangleq \Box\Diamond\langle c(\alpha_r)\rangle$, where

$$c(\alpha_r) \quad \triangleq \quad (\texttt{ncre}[1] = (\ \texttt{EqR}\ , l-1)) \wedge (\texttt{nF}[1] = \ \texttt{neighbor\_l-1}\ \ )$$

After applying all steps of our technique, we obtain a system composed of the transformed version of process $P(1)$ and the network invariant. We described the system $(P(1) \| I_H)$ in smv and used TLV [69] to check the $\Diamond\Box J$ property. It took TLV 35 sec to

prove this property. From the proof we can conclude the $\forall N \geq 2 : (P(1)\|P(2)\|...\|P(N)) \models \Diamond\Box H \rightarrow \Diamond\Box J$.

7.8.2. Leader Election

For the leader election algorithm ([27],p35) a number of processes form an arbitrary connected graph. The purpose of the algorithm is that eventually all processes will agree on the process with the minimum id in the graph to be the leader. In order to achieve that, each process stores a candidate leader and its distance from the leader. Then it reads the values of all its neighbors. If there is a candidate with id smaller than its own leader or with the same id and smaller distance, the process updates its candidate with that value and its distance by incrementing the distance read by 1. The update happens only if the distance of the neighbor's candidate is less than a prespecified constant M, which represents the maximum number of nodes in the graph.

If one of the processes i is initialized with a candidate id v in variable `leader` which is smaller than any of the ids in the graph and any of the other nodes' candidates, then v is going to be stored in other neighbors' `leader` variable and from them again to i. However, each time this "floating" id moves from one node to another, the value of `distance` increases. Therefore, eventually `distance` becomes greater than M for all nodes and the value v does not appear in the `leader` variables of the graph.

For this algorithm we assume that `min_id` is the smallest floating id and `min_dist` is its minimum `distance` value in the graph and we prove that eventually always if a node has `min_id` as a candidate, the distance will be greater than `min_dist`. We use $P(1)$ as the special process, but due to symmetry the proof can be generalized.

This system is not a finite state system because variables storing candidate leaders and distances take values from an infinite domain. Therefore, we abstract those variables to a few values of interest, i.e., $\{\texttt{min\_id}, \texttt{other\_id}\}$ for the candidates and {min_dist, gt_min_dist, any_dist} for the distance, where gt_min_dist denotes greater than min distance. We abstract the part of the graph that does not belong to process 1 and its neighbors and the loop structure that reads the values of all neighbors. After the preprocessing step we are left with a parameterized system that is amenable to our technique. The property $H = \Gamma \wedge \Delta \wedge \forall k \in E(k)$ and J are given by[9]

$$
\begin{aligned}
\Gamma &= \text{True} \\
\Delta &= \quad \wedge \ \texttt{candidate}[1] = \texttt{min\_id} \Rightarrow \texttt{distance}[1] \neq \texttt{any\_dist} \\
&\quad \wedge \ \texttt{leader}[1] = \texttt{min\_id} \Rightarrow \texttt{dis}[1] \neq \texttt{any\_dist} \\
E(k) &= \quad \wedge \ \texttt{candidate}[k] = \texttt{min\_id} \Rightarrow \texttt{distance}[k] \neq \texttt{any\_dist} \\
&\quad \wedge \ \texttt{leader}[k] = \texttt{min\_id} \Rightarrow \texttt{dis}[k] \neq \texttt{any\_dist}
\end{aligned}
$$

$$
\begin{aligned}
J = \quad & \wedge \ \texttt{candidate}[1] = \texttt{min\_id} \Rightarrow \texttt{distance}[1] = \texttt{gt\_min\_dist} \\
& \wedge \ \texttt{leader}[1] = \texttt{min\_id} \Rightarrow \texttt{dis}[1] = \texttt{gt\_min\_dist}
\end{aligned}
$$

Each process executes a low-atomicity algorithm with single-writer, multi-reader communication registers. Moreover, the preconditions of the actions depend on the program counter or on local variables. The usage of the program counter makes the system satisfy $\Lambda 3$, as discussed in Section 7.3. The communication register for this algorithm is $\texttt{cr}[j] = (\texttt{leader}[j], \texttt{dis}[j])$. After the application of our technique the system has a fixed

[9]The rest of the conjuncts in the code have to do with the domain of the variables.

number of observable variables and is amenable to control abstraction. Performing control abstraction we obtain a finite-state system with 500,000 states. We specified the system in TLA+. Then we used the TLC model checker [54], which is based on explicit state enumeration, to prove the correctness of the algorithm. TLC took 22 minutes to prove the property $\Box\Diamond J$ of the abstract system.

7.8.3. Coloring

We apply our technique on the self-stabilizing coloring algorithm ([27],p162). The purpose of this algorithm is to assign a color to each process, such that no two neighbor processes have the same color. Each process keeps reading the values of all its neighbors, stores the values of the neighbors with an id greater than its id, and assigns to its `color` variable a color that none of the neighbors with a higher id have.

We want to prove that eventually always a process will have a color that none of its neighbors with higher id numbers have, if eventually always the neighbors with higher ids are silent processes, i.e., the values of their communication registers are constant. We use $P(1)$ as the special process and with data abstraction we make the state space of each process finite. Then we apply our technique and prove the desired property. We specify the system in TLA+. TLC requires 1 minute and the total number of states is 26,496.

In one of the actions of the program, a process makes a call to a function `choose`() with arguments the set of all colors except the colors of its neighbors with higher ids. If no fault occurs or after a process has completed its first actions, it is guaranteed that the argument passed to `choose`() has at least one element. This is because the number of colors is always greater than the number of neighbors of a node. However, if we start in a

state in which the two sets are equal, the argument passed to `choose`() is the empty set. Therefore, it is important that `choose`() is able to return a color if called with the empty set and not crash[10].

7.9. Summary

This article describes an abstraction technique for a class of self-stabilizing algorithms. The abstraction technique is the first that can be used to prove the correctness of low-atomicity, parameterized self-stabilizing systems with an unbounded number of observable variables for each process. The abstract system derived by the proposed approach is relatively small. As the case studies show the abstraction technique is not trivial and can be applied to a number of self-stabilizing algorithms. Moreover, the conditions of completeness of each step are defined.

Not all self-stabilizing algorithms can be directly handled by our abstraction technique. There are algorithms containing variables with infinite domains, loop structures that read the values of all its neighbors, or relations specified on the ids of the processes. Moreover, in many distributed systems a process and its neighbors do not form a closed system, but are part of a larger arbitrary graph. There are ways to make these algorithms amenable to our abstraction technique during the preprocessing step. However, the focus of this work is the abstraction of a parameterized self-stabilizing system with an unbounded number of observable variables for each process. The proposed abstraction technique enables automated verification methods, i.e., model-checking, to check the correctness of a class of self-stabilizing algorithms.

[10]In the paper [28] from which the paragraph in [27] is motivated, the authors mention that.

CHAPTER 8

Conclusions

8.1. Summary

Our work targeted the system-level synthesis and verification of digital integrated circuits. Several algorithms were developed that can improve the quality of synthesis. More specifically, a dynamic programming algorithm was presented that produces a schedule with reduced energy consumption for a system-level pipelined streaming application. Moreover, a retiming algorithm was developed for applications that can be represented as Synchronous Data Flow graphs. The algorithm produces a schedule with the minimum iteration latency. Furthermore, the optimization power of the synthesis operations was explored and a sequence of synthesis operations was proposed that can perform any sequential transformation.

System-level and high-level synthesis will be widely adopted only if the correctness of their results can be checked. However, in most cases the verification problem is harder than synthesis. We proposed several approaches to speed up verification. The first is based on recording the history of the transformation during synthesis. Even though by using this approach the speedup can be significant in some cases, the most efficient way to improve verification runtime is by using abstraction. We presented several abstraction methods. The first can be used for computational intensive system-level applications, in which a bit accurate description is not necessary. In that case we represent all data types

in the word-level. The second enables the use of automated tools for the verification of parameterized self-stabilizing systems. The technique abstracts an arbitrary number of processes to a small finite state system.

8.2. Future Work

There are still many interesting problems in this field. A very important problem is to define a framework for sequential synthesis that is verifiable by efficient sequential equivalence checking procedures. We saw that the sequence of retiming and resynthesis operations is a powerful synthesis paradigm, but has a number of limitations. Moreover, it does not guarantee that the result of synthesis can be checked by any method other than model checking. Therefore, it is useful to consider whether there exists a synthesis framework that is more powerful than the RnR sequence and whose results can be verified by a more efficient verification method.

Transactional memories is another interesting research field. For deriving the optimization power of retiming and resynthesis we assumed that the functions between the blocks of register are combinational. The reason is that the evaluation of these functions is an atomic action by the system, i.e., an action that is indivisible and happens instantly before the rising edge of the clock. Transactional memories are an architectural implementation that can guarantee atomicity for actions performing memory operations. It would useful to consider the optimization power of retiming as a scheduling operation when transactional memories are involved.

In Chapters 2 and 3 we saw how synthesis operations applied to the structural representation of the design defined the scheduling and communication between processes.

In most cases the structural representation is more complete than the behavioral representation. For example, control units could be missing in the behavioral representation. An investigation on the effect and optimization power of other structural synthesis operations would be very interesting. Since the structural representation is more complete, the quality of the synthesis result could be improved.

Finally, the type of abstraction used during verification should be decided by the abstraction used during synthesis. It is interesting to see how we can pass information stored during the synthesis phase for the verification procedure, such that the complexity of the verification is reduced, while the possibility of a false positive answer is eliminated. This requires that the information passed can be independently checked during verification. Research in this area can tackle the kind of information and data structures that are needed for this approach.

References

[1] Abadi, M., and Lamport, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (1991).

[2] Adé, M., Lauwereins, R., and Peperstraete, J. A. Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC '97: Proceedings of the 34th annual conference on Design Automation* (New York, NY, USA, 1997), ACM, pp. 64–69.

[3] Arora, A., and Gouda, M. Distributed reset. *IEEE Transactions on Computers 43*, 9 (1994).

[4] Ashar, P., Gupta, A., and Malik, S. Using complete-1-distinguishability for fsm equivalence checking. *ACM Trans. Des. Autom. Electron. Syst. 6*, 4 (2001), 569–590.

[5] Aziz, A., Singhal, V., and Brayton, R. K. Verifying Interacting Finite State Machines. Tech. Rep. UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.

[6] Bakhshi, R., and Gurov, D. Verification of peer-to-peer algorithms: A case study. *Electronic Notes in Theoretical Computer Science 181* (2007), 35–47.

[7] Balcázar, J. L., Lozano, A., and Torán, J. The complexity of algorithmic problems on succinct instances. *Computer science: research and applications* (1992), 351–377.

[8] Barrett, C. W., Dill, D. L., and Stump, A. A framework for cooperating decision procedures. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction* (London, UK, 2000), Springer-Verlag, pp. 79–98.

[9] Baukus, K., Lakhnech, Y., and Stahl, K. Verification of parameterized protocols. *Journal of Universal Computer Science 7*, 2 (2001), 141–158.

[10] Baumgartner, J., Kuehlmann, A., and Abraham, J. A. Property checking via structural analysis. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification* (London, UK, 2002), Springer-Verlag, pp. 151–165.

[11] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, Release, 70930,http://www.eecs.berkeley.edu/˜alanmi/abc/.

[12] Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems 21*, 2 (1999), 151–166.

[13] Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. Symbolic model checking using sat procedures instead of bdds. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation* (New York, NY, USA, 1999), ACM, pp. 317–320.

[14] Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A. L., Somenzi, F., and et. al., A. A. Vis: A system for verification and synthesis. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification* (London, UK, 1996), Springer-Verlag, pp. 428–432.

[15] Burd, T. D., and Brodersen, R. W. Design issues for dynamic voltage scaling.

In *ISLPED '00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2000), ACM, pp. 9–14.

[16] Burns, J. E., Gouda, M. G., and Miller, R. E. Stabilization and pseudo-stabilization. *Distributed Computing 7*, 1 (1993), 35–42.

[17] Calypto Design Systems. Navigating the System to RTL Continuum, http://www.calypto.com/whitepapers.php.

[18] Cho, H., Hachtel, G. D., and Somenzi, F. Redundancy identification and removal based on implicit state enumeration. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors* (Washington, DC, USA, 1991), IEEE Computer Society, pp. 77–80.

[19] Clarke, E., Kroening, D., and Yorav, K. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC '03: Proceedings of the 40th conference on Design automation* (New York, NY, USA, 2003), ACM, pp. 368–371.

[20] Clarke, E., Kroening, D., and Yorav, K. Behavioral consistency of c and verilog programs using bounded model checking. In *CMU-CS-03-126, Carnegie Mellon University, School of Computer Science* (2003).

[21] Clarke, E., Talupur, M., and Veith, H. Environment abstraction for parameterized verification. In *VMCAI 2006: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation* (London, UK, 2006), Springer-Verlag, pp. 126–141.

[22] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*

(London, UK, 1982), Springer-Verlag, pp. 52–71.

[23] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 1999.

[24] Currie, D. W., Hu, A. J., and Rajan, S. Automatic formal verification of dsp software. In *DAC '00: Proceedings of the 37th conference on Design automation* (New York, NY, USA, 2000), ACM, pp. 130–135.

[25] De Micheli, G. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE TCAD 10*, 1 (Jan. 1991), 63–73.

[26] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[27] Dolev, S. *Self-Stabilization*. The MIT Press, 2000.

[28] Dolev, S., and Herman, T. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science 1997*, 4 (December 1997).

[29] Eles, P., Peng, Z., Kuchcinski, K., and Doboli, A. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems 2*, 1 (1997), 5–32.

[30] Emerson, E. A., and Kahlon, V. Reducing model checking of the many to the few. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction* (London, UK, 2000), Springer-Verlag, pp. 236–254.

[31] Estlick, M., Leeser, M., Theiler, J., and Szymanski, J. J. Algorithmic transformations in the implementation of k- means clustering on reconfigurable hardware. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays* (New York, NY, USA, 2001), ACM, pp. 103–110.

[32] Even, G., Spillinger, I. Y., and Stok, L. Retiming Revisited and Reversed. *IEEE TCAD 15*, 3 (Mar. 1996), 348–357.

[33] Fang, Y., Piterman, N., Pnueli, A., and Zuck, L. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer (STTT) 8*, 3 (2006), 261–279.

[34] Gouda, M. G., and Multari, N. J. Stabilizing communication protocols. *IEEE Transactions on Computers 40*, 4 (1991), 448–458.

[35] Govindarajan, R., and Gao, G. R. Rate-optimal schedule for multi-rate dsp computations. *Journal of VLSI Signal Processing Systems 9*, 3 (1995), 211–232.

[36] Hasteer, G., Mathur, A., and Banerjee, P. Efficient equivalence checking of multi-phase designs using retiming. *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on* (8-12 Nov 1998), 557–562.

[37] Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., and Bose, P. Microarchitectural techniques for power gating of execution units. In *ISLPED '04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2004), ACM, pp. 32–37.

[38] Im, C., Kim, H., and Ha, S. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2001), ACM, pp. 34–39.

[39] Jiang, J.-H. R., and Brayton, R. K. Retiming and resynthesis: A complexity perspective. *Computer-Aided Design of Integrated Circuits and Systems, IEEE*

Transactions on 25, 12 (Dec. 2006), 2674–2686.

[40] Jiang, J.-H. R., and Hung, W.-L. Inductive equivalence checking under retiming and resynthesis. In *ICCAD '07: Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)* (2007).

[41] Joisha, P., and Banerjee, P. The MAGICA type inference engine for MATLAB. In *Proceedings of the 12th International Conference on Compiler Construction* (2003).

[42] Joisha, P. G., and Banerjee, P. PARADIGM (version 2.0): A new HPF compilation system. In *IPPS '99/SPDP '99: Proceedings of the International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 609–615.

[43] Kesten, Y., and Pnueli, A. Control and data abstraction: the cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer (STTT) 2*, 4 (2000).

[44] Kesten, Y., and Pnueli, A. Verification by augmented finitary abstraction. *Information and Computation 163*, 1 (2000), 203–243.

[45] Kesten, Y., Pnueli, A., Shahar, E., and Zuck, L. D. Network invariants in action. In *CONCUR '02: Proceedings of the International Conference on Concurrency Theory* (London, UK, 2002), Springer-Verlag, pp. 101–115.

[46] Keutzer, K., Newton, A., Rabaey, J., and Sangiovanni-Vincentelli, A. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 19*, 12 (Dec 2000), 1523–1543.

[47] Khouri, K. S., and Jha, N. K. Leakage power analysis and reduction during

behavioral synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10* (December 2002), 876–885.

[48] Kim, C., and Roy, K. Dynamic vth scaling scheme for active leakage power reduction. In *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2002), IEEE Computer Society, p. 163.

[49] Kropf, T. *Introduction to Formal Hardware Verification*. Springer, 1999.

[50] Kuehlmann, A., and Baumgartner, J. Transformation-based verification using generalized retiming. In *CAV '01: Proceedings of the International Conference on Computer Aided Verification* (London, UK, 2001), Springer-Verlag, pp. 104–117.

[51] Kulkarni, S. S., and Arumugam, U. Collision-free communication in sensor networks. In *Self-Stabilizing Systems: 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003. Proceedings* (2003), Springer Berlin / Heidelberg.

[52] Kurshan, R. P., and McMillan, K. L. A structural induction theorem for processes. *Information and Computation 117*, 1 (1995), 1–11.

[53] Lagnese, E., and Thomas, D. Architectural partitioning for system level synthesis of integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 10*, 7 (Jul 1991), 847–860.

[54] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[55] Lee, E. A., and Messerschmitt, D. G. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers 36*, 1 (1987), 24–35.

[56] Leiserson, C. E., and Saxe, J. B. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems 1*, 1 (Spring 1983), 41–67.

[57] Leiserson, C. E., and Saxe, J. B. Retiming synchronous circuitry. *Algorithmica 6*, 1 (1991), 5–35.

[58] Lesens, D., Halbwachs, N., and Raymond, P. Automatic verification of parameterized linear networks of processes. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 346–357.

[59] Lin, B., Touati, H. J., and Newton, A. R. Don't Care Minimization of Multi-level Sequential Logic Networks. In *ICCAD* (Nov. 1990), pp. 414–417.

[60] Lin, C., and Zhou, H. Optimal wire retiming without binary search. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 452–458.

[61] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[62] Malik, S. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, Nov. 1990. Memorandum No. UCB/ERL M90/115.

[63] Malik, S., Sentovich, E., Brayton, R., and Sangiovanni-Vincentelli, A. Retiming and resynthesis: optimizing sequential networks with combinational techniques. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 10*, 1 (Jan 1991), 74–84.

[64] McMillan, K. L. *Symbolic Model Checking*. Springer, 1993.

[65] McMillan, K. L. Applying sat methods in unbounded symbolic model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification* (London, UK, 2002), Springer-Verlag, pp. 250–264.

[66] Mishchenko, A., and Brayton, R. K. Recording synthesis history for sequential verification. In *IWLS* (2008).

[67] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation* (New York, NY, USA, 2001), ACM, pp. 530–535.

[68] Murthy, P., Bhattacharyya, S., and Lee, E. Minimizing memory requirements for chain-structured synchronous dataflow programs. In *ICASSP-94: IEEE International Conference on Acoustics, Speech, and Signal Processing* (1994), vol. 2, pp. 453–456.

[69] NYU. Temporal Logic Verifier, 2008,http://www.cs.nyu.edu/acsys/tlv/index.html.

[70] O'Neil, T. W., and Sha, E. Retiming synchronous data-flow graphs to reduce execution time. *IEEE Transaction on Signal Processing 49*, 10 (2001), 2397–2407.

[71] OPEN SYSTEMC INITIATIVE. SystemC, http://www.systemc.org.

[72] Owre, S., Rushby, J. M., and Shankar, N. Pvs: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction* (London, UK, 1992), Springer-Verlag, pp. 748–752.

[73] Pino, J. L., Bhattacharyya, S. S., and Lee, E. A. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Tech. Rep. UCB/ERL-95-36, Electronics Research Laboratory, University of California at Berkeley, May 1995.

[74] Pnueli, A., Xu, J., and Zuck, L. Liveness with (0,1,infinity)-counter abstraction. In *CAV '02: Proceedings of the International Conference on Computer Aided Verification* (London, UK, 2002), Springer-Verlag, pp. 107–122.

[75] Rabaey, J. M., Chandrakasan, A., and Nikolic, B. *Digital Integrated Circuits*. Prentice Hall, 2003.

[76] Ranjan, R. K., Singhal, V., Somenzi, F., and Brayton, R. K. On the optimization power of retiming and resynthesis transformations. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design* (New York, NY, USA, 1998), ACM, pp. 402–407.

[77] Ritz, S., Pankert, M., and Meyr, V. Z. H. Optimum vectorization of scalable synchronous dataflow graphs. In *International Conference on Application-Specific Array Processors* (1993), pp. 285–296.

[78] Singhal, V., Malik, S., and Brayton, R. K. The Case for Retiming with Explicit Reset Circuitry. In *ICCAD* (Nov. 1996), pp. 618–625.

[79] Stoffel, D., and Kunz, W. Record and Play: a Structural Fixed Point Iteration for Sequential Circuit Verification. In *ICCAD* (1997), pp. 394–399.

[80] Stroustrup, B. *The C++ Programming Language*. Addison Wesley, 200.

[81] Stump, A., Barrett, C. W., and Dill, D. L. Cvc: A cooperating validity checker. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification* (London, UK, 2002), Springer-Verlag, pp. 500–504.

[82] Theocharides, T., Michael, M., Polycarpou, M., and Dingankar, A. A novel system-level on-chip resource allocation method for manycore architectures. *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual* (April

2008), 99–104.

[83] Touati, H. J., and Brayton, R. K. Computing the Initial States of Retimed Circuits. *IEEE TCAD 12*, 1 (Jan. 1993), 157–162.

[84] van Eijk, C. A. J. Sequential Equivalence Checking without State Space Traversal. In *DATE* (Paris, France, 1998), pp. 618–623.

[85] van Eijk, C. A. J. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 19*, 7 (2000), 814–819.

[86] Whittlesey-Harris, R. S., and Nesterenko, M. Fault-tolerance verification of the fluids and combustion facility of the international space station. In *ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 5.

[87] WOLFRAM. Mathematica Package, http://www.wolfram.com.

[88] Zhou, H. A new efficient algorithm derived by formal manipulation. In *IWLS '04: Workshop Notes of International Workshop on Logic Synthesis* (2004).

[89] Zhou, H., Singhal, V., and Aziz, A. How powerful is retiming? In *IWLS* (1998).

[90] Zivojnoivic, V., Ritz, S., and Meyr, H. Retiming of dsp programs for optimum vectorization. In *ICASSP 94: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (1994).

[91] Zivojnovic, V., and Schoenen, R. On retiming of multirate dsp algorithms. In *ICASSP 96: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (1996).