Formal Verification throughout the Development of Robust Systems

Niels Thole
Group of Reliable Embedded Systems
University of Bremen

Dissertation
for the acquisition of the doctorate in engineering
— Dr.-Ing. —

Primary Reviewer
Prof. Dr. Görschwin Fey

Secondary Reviewer
Prof. Dr. Alberto Garcia-Ortiz

October 7, 2016

Acknowledgments

I want to thank everyone who supported me during my time as a PhD student.

First, I want to thank Görschwin Fey, my adviser and first reviewer. He greatly supported me over my four years at the University of Bremen regarding almost all aspects of my thesis. It was also great to work with other co-authors of shared articles. Heinz Riener, Alberto Garcia-Ortiz, and Lorena Anghel were all very supportive when we worked together.

The Group of Reliable and Embedded Systems offered a great work environment. I enjoyed the informal atmosphere and the productive discussions with everyone.

Finally, I want to thank my parents, my sister, and my wife for their moral support.

Contents

Chapter 1

Introduction

1.1 Motivation

New technologies regularly facilitate smaller and faster transistors which in turn enables more powerful systems that use more transistors. While some focus on manufacturing or materials, others use new concepts like Memristors [28] to enable more functionality on a single circuit. According to Moore's Law [40], the number of transistors in an integrated circuit doubles once every 18 months. While Moore's Law is only an estimation, it remained true from 1975 until 2012. Nowadays, the rate has slowed, but according to Intel [1], the number of transistors will still double approximately every two and a half years until 2017. As the size of transistors decreases, the size of the devices remains the same or even shrinks.

The increase of transistors allows more powerful systems in almost every area of our lives and increase our quality of life. In addition, the size of the systems keeps decreasing allowing powerful computers that require less space and enables devices like smart phones or advanced infotainment systems in cars. As the devices become smaller, they require less electrical current to work and thus safe energy which can be used to provide more computational power to battery driven devices for more time. And the systems not only become more powerful, smaller, and energy efficient, they even become cheaper as developers often use existing off-the-shelf hard- or software instead of developing their own specialized implementation.

The high number and small size of transistors provide a foundation for powerful and energy efficient systems. However, this development not only provides advantages, but leads to new challenges as well. As systems become more complex, bugs during the development become more likely and smaller parts are more susceptible to faults during production or from external sources like cosmic radiation.

In short, systems must be constructed in a way that prevents *errors*. Errors describe visible behavior of the system that is different from the specification. Examples include freezing applications, erroneous output signals, or delayed output values. The two major reasons for errors are bugs and faults.

Bugs describe differences between the implementation and the specification. They can be caused by a mistake of the developer or can be inherited from used

subsystems. Examples for bugs are implementations of functions that return wrong results under certain circumstances. These bugs do not necessarily lead to errors. It is possible that a bug is avoided as the corner case that would be affected by the bug cannot happen in the system or the buggy output values are checked and corrected within the system before they become visible.

To detect and remove bugs, testing or formal verification is applied. Both approaches aim to detect possible executions of the system or parts of the system under which a bug changes the visible or invisible behavior of the system. When a buggy execution is detected, the corresponding bug needs to be localized within the system and corrected. This can be a manual effort or can be done with the help of tools that analyze the buggy execution trace and detect likely locations of the bug.

Another cause for errors are *faults*. Faults are defects within the system that are caused by external effects and can change the behavior of the system. Faults can be permanent or transient. Permanent faults can have different causes, e.g., process variation during production, aging, or radiation, and cause a permanent change in the system that could cause errors. Just like permanent faults, transient faults change the behavior of the system. However, this change only lasts for a short time and disappears afterwards. Transient faults are usually caused by radiation that causes ionized particles to hit the system and can interact with the electronics.

As faults are not part of the implementation, they cannot be removed. Instead, a system needs to be *hardened* against the relevant faults. Hardening is a process to modify the system such that it is *robust* against certain faults. If a system is robust against a fault, it can prevent that fault from causing an error. When implementing a robust system, it needs to be shown that

1. the system fulfills its specifications and
2. the system is robust against the considered faults.

During the implementation of a robust system, development usually starts with a non-robust model that is modified during multiple iterations. At some iterations, the system is hardened against faults. To guarantee that the hardening is successful, it needs to be shown, that the hardened system still fulfills the requirements.

This can be done by showing that the hardened system behaves equivalently to the previous non-robust iteration. However, proving equivalence can be difficult, as the implementation of the system can change significantly even though the system still acts the same from an external view. In addition, the interface of the system can change during an iteration. In that case, it needs to be shown that an execution on the previous system outputs the same values as a corresponding execution on the new iteration of the system, even though the inputs differ based upon the new interface. The modifications during the iteration could even change the behavior of the system for non-reachable states. As these states cannot be reached due to the implementation of the system, the knowledge about reachable and non-reachable states needs to be generated by or provided to a tool that is meant to show equivalence. Otherwise, these non-reachable states could falsely be returned as counterexamples to disprove equivalence. As such, an algorithm to prove equivalence needs to handle these challenges to reach a correct decision about equivalence.

Furthermore, it needs to be shown that the hardening provides the robustness that it is meant to and the hardened system is robust against the considered faults. There are different approaches to verify this. Usually, faults are injected into the system. Depending on the model and abstraction level, this can be realized in different ways. Testing or formal verification shows that the injected faults do not change the output behavior of the system. Complications can arise when the fault affects different internal parts of the system and changes their behavior as the fault propagates through the system. Modeling these reconvergences in formal approaches is especially complex as all possible behavior needs to be included in the model.

Hardening can be done on different abstraction levels. For example, error correcting codes can already be introduced on *Electronic System Level* (ESL). Providing additional hardware to provide some redundancy is usually done in a *Hardware Description Language* (HDL). Another option is to use bigger transistors at critical locations as these are less likely to be affected by transient faults due to radiation, which is done at the transistor level. As hardening can be done on almost every abstraction level, techniques to verify that hardening is implemented correctly is also needed on these different levels.

1.2 Contributions

This thesis contains two major contributions: an equivalence checker for C++ classes [59, 60] and a robustness checker to verify if a circuit is robust against a *Single Event Transient* (SET) [58, 57].

The equivalence checker can be used to verify if a hardened and a non-hardened system, described on ESL, behave equivalently in the absence of faults and thus show that the hardened version fulfills its functional specifications. Other applications, like verifying the equivalence of different versions of a system during an iterative design process are also possible.

The equivalence checker uses an inductive approach to prove equivalence of two hardware models, given as C++ classes. Using an inductive approach avoids unrolling which requires high effort when long execution paths are considered. However, while unrolling ensures that only reachable states are considered, this information needs to be given or generated for an inductive approach. For this reason, we use a candidate invariant, i.e., an approximation of reachable and corresponding variable assignments of the two classes. This candidate invariant is given by the developer. By providing a good candidate invariant, the developer is able to provide his knowledge about the internal structure of the system and the correspondence between the two checked models to significantly speed up the equivalence check. If the current state of the two classes corresponds to the candidate invariant and both classes execute the same functions, they need to return equivalent output and reach variable assignments that are described by the candidate invariant as well. If this can be shown by an underlying model checker, the models are equivalent. Otherwise, the model checker returns a counter example. If the counter example describes reachable and non-equivalent behavior, we have proven that the models are not equivalent. If we could neither prove equivalence nor non-equivalence, we use the counterexample to refine the candidate invariant. As both classes usually consist of a huge amount of states, blocking individual states that are provided by counterexamples does usually

not suffice to finish the decision within feasible time. To handle and block multiple states at the same time, multiple heuristics are used to generalize the counterexample and extract additional information about non-reachable states. This process is repeated until equivalence is decided.

While the initial candidate invariant allows a designer to provide his knowledge about the system to the equivalence checker, the candidate invariant can also be generated by third-party tools or set to "true", which is the coarsest possible overapproximation and would describe all pairs of states of the two models. The initial candidate invariant is one of the advantages of our approach and enables a developer to provide a significant speed up by providing additional knowledge about the models.

We will describe two different versions of the equivalence checker, NSMC [59] and EASY [60], that use different approaches to learn and adjust the candidate invariant. In the experiments, we show the performance and scalability of both approaches on some examples and show that a good hypothesis enables a decision within a feasible time, even when considering complex examples.

The robustness checker verifies if a given gate level circuit is robust against an SET. An SET describes that the output of a gate is negated for a short duration within a single clock cycle. This can be used to prove that the hardening of a system was successful and SETs do not affect the output of the system.

When we prove robustness of the circuit, we consider most effects that are relevant for the behavior of the physical circuit. We include logical, timing, and electrical masking in our model. In addition, we consider variability, meaning that the behavior of the gates is uncertain to a very small degree due to process variation. In this aspect, our approach is unique, as at this time no other formal robustness checker considers variability. We describe variability by having variable delays for each gate. Considering all these effects on the analog signals within a circuit would require an extremely complicated model which is not feasible. Thus, we need to abstract some details while still providing significant results. For this reason, we did some conservative adjustments and use three-valued logic for the signals within the circuit. Three valued-logic allows us to consider a signal as unknown during specific times. We use these unknown values to describe rising and falling flanks of a signal as the binary interpretation is uncertain during that time. When the output of a gate becomes uncertain depending on variability the output is also considered as unknown.

If the robustness checker decides that the circuit is robust against an SET, this decision can be transferred to the final system as all our abstractions are conservative. Otherwise, if the algorithm returns a counterexample that disproves robustness of our model, it is possible that the counterexample cannot be applied to the final system and the system is robust after all as our model is too abstract in this case. Further analysis, e.g., spice simulation, can be used to verify whether a generated counterexample is real or spurious.

The first version [58] of the robustness checker decides robustness by using a monolithic approach. The behavior of the circuit under the SET is encoded as a SAT formula. If the formula is satisfiable, the satisfying assignment provides a counterexample against robustness. Otherwise, the circuit is robust against the SET.

As signals can become extremely complex when the SET splits and re-converges, the resulting SAT formula from our monolithic approach became correspondingly complex. We used another approach to tackle the problem [57].

This time, we use a hybrid approach that partitions the circuit into a front and a back area. Only the front area is used to generate the SAT formula and generated counterexamples are further verified by using simulation. When the SAT solver generates a counterexample that does not affect the primary outputs of the circuit, the detected counterexample is spurious and does not disprove robustness. Instead, the SAT formula is modified to block the detected and similar assignments and the SAT solver is run again until either the SAT formula becomes unsatisfiable or a real counterexample is found.

The experiments with the robustness checkers show that both approaches provide a significant speed up compared to spice simulations as spice simulation to consider all possible variability takes hours, even on the smallest circuit c17 with 5 gates of the ISCAS-85 benchmark. In comparison, our checkers require only seconds. Furthermore, we will show that the second version can decide robustness significantly faster and provides an average speed up of 748 compared to the first version.

1.3 Structure

The following Chapter 2 describes a common design flow for robust systems. We highlight how verification is used during the design process to ensure that the system fulfills its requirements during the iterations of development. In addition, we present relevant fault models that should be considered when developing a robust system as these models describe typical real world faults. Methods to harden a system against these faults are presented as well as approaches to verify the robustness of the system. In that chapter, we will also provide insight into related work to our contributions and discuss the differences.

Chapter 3 shows our contributions for equivalence checking. After introducing the required preliminaries and used data structures for our approaches, the initial equivalence checker NSMC is presented. Next, the algorithm EASY is shown. Experiments show the performance of both algorithms and compare them.

The robustness checkers are presented in Chapter 4. After providing some preliminaries, the monolithic robustness checker is described. The hybrid robustness checker is shown as well. Experiments validate the correctness of the algorithms, present their runtime, and show the effects of different parameters.

The final Chapter 5 gives an outlook for future work. Further expansions or optimizations to the presented algorithms are sketched and ideas to combine the equivalence checker and the robustness checker are given.

Chapter 2

Developing Robust Systems

In this chapter, we describe a common design process to develop robust systems. We start with a small example of the process. The general design flow that originates from an informal description and continuously decreases the abstraction level until the system is completed as a chip or similar hardware element is shown. Next, we describe some fault models that can be used during the development process. These fault models describe common faults that the system needs to be robust against. Some general methods to provide this robustness are introduced afterwards. We conclude this chapter with methods to analyze and verify robustness.

2.1 Example: Developing a Counter

Let us consider an example for the development flow. In this example, the flow will be applied to develop a counter. The process is sketched in Figure 2.1.

The process starts by defining the requirements of the system. In our case, the counter should count from 0 to 3 and be robust against single SETs. For this example, we will only focus on a small number of requirements. Usually, additional requirements focus on different aspects of the system, e.g., required space and power, used inputs and outputs, or the frequency of the system. Even this small example shows some of the problems of natural language as the behavior of the system is not completely defined. What happens when the counter reaches 3? Does it need to be robust against the first SET or does it need to handle SETs as long as there appears only one SET at a time?

Based upon the requirements, the system is defined by using a modeling language. In our example, we describe the system with a class diagram in UML. The diagram shows a class counter, that contains an integer value and a method `countUp`. UML can be used to provide more details about the behavior of the system by using Behavior Diagrams like Use Case Diagrams, but we will not use these in our example.

When transforming the requirements into UML models, the developers need to ensure that the models include the requirements. In our case, the requirement of being able to count is meant to be realized by the method `countUp`. The robustness is not yet modeled at this level.

Next, the UML description is used to implement the system at ESL. We

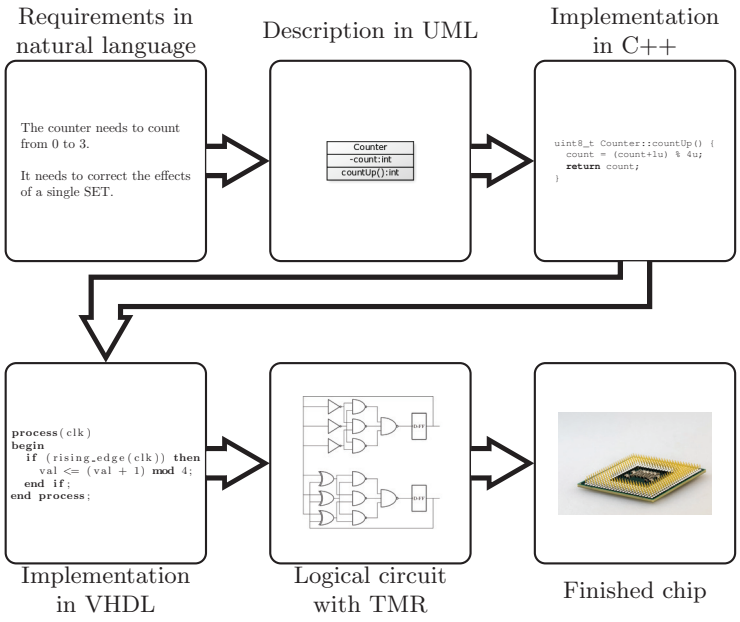


Figure 2.1: Development flow of a counter

use the class diagram as model for a C++ class `Counter` and implement the method `countUp`. The method increases the value by 1 and uses the modulo operation to ensure that it remains between 0 and 3. With this implementation we also defined what happens when the counter reaches its maximum value of 3 as it is reset to 0 with the next execution of `countUp`. In addition, we further specified the integer type. We use an unsigned 8-bit integer as this is obviously sufficient to represent the numbers between 0 and 3. It can easily and automatically be shown that the implementation in C++ is consistent with the UML model since both use the same names for variables and methods.

Based upon the C++ implementation, we implement the system with an HDL. In our case, we use VHDL. The shown process describes that the modulo operation is executed whenever the clock rises, increasing the stored value every clock cycle.

To check that both models of the counter are equivalent, we need to define the correspondences between the models. Since the circuit counts up every timestep, the method `countUp` corresponds to a rising edge of the clock. The current value of the counter, which is also the return value of `countUp`, is stored in signal `val` and is output in the primary outputs of the circuit. After the correspondence between the primary outputs of the circuit and the return value of `countUp` has been defined, we can use an equivalence checker to verify that both models behave equivalently. Once the equivalence is shown, the logical circuit is generated from the VHDL code.

So far, we did not include the requirement of robustness since an SET does affect the circuit of the system. Since we are at gate level now, an SET can easily be modeled, so we will harden the circuit in the next step. We use TMR to provide robustness.

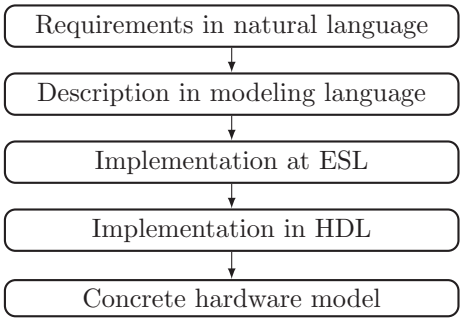


Figure 2.2: Common Design Flow

After hardening the circuit, we need to verify that the hardened and the unhardened circuits behave equivalently in the absence of faults and that the hardened circuit is robust against SETs. The equivalence is easily and automatically shown. However, the robustness check will show that the circuit is not robust against all possible SETs. If an SET affects a gate within the original part of the adder, i.e., a `not`- or an `xor`-gate, the fault will be corrected, but when the voter, i.e., a `nand`-gate, is affected, the SET could result in an error. We will not modify the circuit further, but will handle this problem at the next abstraction level instead.

In the end, the circuit will be turned into a hardware chip. We need to place and route the gates and include electrical components like batteries to create the chip of the counter. To handle the vulnerability against SETs within the voter, we decide to use bigger transistors for the `nand`-gates to harden these against SETs.

After verifying that the chip behaves equivalently to the final circuit at gate level and is robust against SETs, we can ensure that we have created a chip that fulfills the initial requirements.

2.2 Design Flow

The design flow of robust systems is similar to the flow for regular systems. We present a common development flow [49] that is sketched in Figure 2.2. The flow starts with the highly abstract level of natural language and includes additional details in multiple abstraction levels until the concrete system is described on hardware level. Depending on the needs of the customer and developer this flow can be modified to include additional abstraction levels or skip certain levels.

The development begins with a description of the system in natural language. This description contains information about the features of the system. This description is usually created by the developers together with the client who orders the system. Common practices exist to structure this process, including but not limited to formulating requirements [2]. This description is meant to be understood by humans and can usually only be understood to a limited degree by computers. Even though the understanding is limited, different techniques [12, 26] exist to automatically interpret the natural language and can support the developers in further steps of the development.

After the system has been defined in natural language the description is modeled by using modeling languages like UML. The resulting model of the system describes the structure and main components of the system but omits most finer details like the specific functionality of methods or processes. Unlike natural language, the semantics of a modeling language provide increased precision and less variation. However, the semantic is not unique as some details of the language are not specified. Verification is already possible at this level and requirements to the described structure can be verified. As the model is still an abstract description of the system, requirements that refer to finer details of the system need to be weakened and cannot be checked completely or are impossible to be checked at this abstraction level.

Based upon the model in a modeling language, a next model is implemented on ESL [5]. An ESL model is implemented with a high level programming language like Java, C++, or SystemC. Most functions of the system can be implemented at this level, but most programming languages on ESL omit details like timing or required energy. In this regard SystemC is an exception as it enables the developer to include more technical and hardware-specific details. Thus, SystemC decreases the distance to further abstraction levels. As the ESL model is implemented in a programming language, the resulting model can be compiled and executed. On ESL, functions can be implemented in a way that is not possible in the final hardware system. Implementing the functionality like this quickly may still be useful, as the system can be executed early on and thus can be tested and verified. In further iterations on ESL, the functionality can be adjusted to be closer to a hardware implementation.

The next abstraction level is the *Register Transfer Level* (RTL) [61]. The RTL describes how the contents of the registers change during the execution of the system. Unlike ESL, descriptions on RTL are inherently parallel and much closer to the real hardware. Some specifics can already be estimated at this level. Nevertheless, RTL is not precise, as some effects like the exact position of gates and the resulting delays caused by the interconnects are not modeled on RTL. To define the RTL model, an HDL like VHDL or Verilog is used.

A *gate level* model can be extracted from the HDL description. This process is called *Logic Synthesis* and is usually done automatically by tools like Design Compiler from Synopsys. The gate level describes the interconnects between logic gates and registers within the electrical circuits of the system.

The final step is *Place and Route* [31], where the hardware elements are put to specific locations on the chip and are connected. During the Placement, a developer decides the exact locations of the gates within the possible limited space. During the Routing, the wires that connect the gates are defined. When this step is done, the layout of the system is defined. This final model includes all details of the system and can be used for final verification and as blueprint to produce the system as hardware after the verification is done.

2.3 Verification during the Design Process

Testing or formal verification are methods to verify if the current description of the system fulfills given requirements [22]. Both can be done on almost every abstraction level. The requirements that can be checked depend on the considered level. If the level of detail is not sufficient for a requirement, it needs

to be checked after the required details have been included or can be checked under certain assumptions.

Testing [44] usually runs a specific set of test cases on a system. Each test case defines input values and expected output values. When a test case is executed, the inputs of the system are set according to the definition of the test case. After the system has generated output values, the real output values are compared to the expected values. If the real values equal the expected values, the test case holds. Otherwise, the test case fails. When all test cases hold, the system is expected to fulfill its requirements. However, exhaustive testing, i.e., testing that considers all possible inputs, is usually not feasible and therefore testing cannot prove the absence of errors. Nevertheless, testing can be used to detect errors. In addition, when the executed tests fulfill certain coverages, it can be assumed that the requirements hold in most situations.

On the other hand, *Formal Verification* [46] is a method to formally prove that the system fulfills certain properties, e.g., its requirements. For this prove, the system and the property are transformed into mathematical models or formulas, e.g., finite state machines and boolean logic formulas. On the generated model, a mathematical proof is done to show that the model always fulfills the property. If formal verification is done successfully, it proves that the system fulfills the requirements under all possible input values. If a property is not fulfilled, formal verification usually generates a counterexample. The counterexample describes an execution of the system that does not fulfill the requirement and can be used to correct the system.

Equivalence checking is a special case of formal verification and verifies whether two models behave equivalently and are functionally equivalent. This method can be used to ensure that the functionality of the system did not change during following iterations as these iterations are usually meant to bring the system closer to the final product but are not meant to change the behavior. In addition, when the previous iteration fulfilled the requirements of the system and is equivalent to the next iteration, the next iteration fulfills the requirements as well.

Two systems are functionally equivalent *if and only if* (iff) they generate corresponding series of output values under corresponding series of input values. If the interfaces of of both systems are the same, this correspondence is defined straight forward as equality, i.e., inputs or outputs with the same name need to produce equal values. However, if the interfaces are different, the correspondence needs to be specifically defined. For example, in an early version at ESL a number could be output as an integer. In further versions, this integer could be split into multiple bits to describe the number. The correspondence can become even more complicated if different abstraction levels are involved. Inputs to a circuit need to be matched to method calls at ESL and it needs to be defined where the return value of that method can be found within the circuit.

Among other uses, equivalence checking can be used to verify if the hardening that has been done within an iteration did not change the nominal behavior of the system.

Today, RTL-to-RTL equivalence checking is typically available in commercial *Electronic Design Automation* (EDA) tools.

An approach to ESL-to-ESL equivalence checking is described in [6]. They prove equivalence between an original program and the version of that program that is optimized by a possibly not trustworthy compiler. To prove equivalence,

the original and the optimized version of the program are turned into the Petri net based Representation for Embedded Systems PRES+. The equivalence between the PRES+ models is shown by verifying that for each path in one model, a corresponding path exists in the second model. While their model considers paths of a program, our equivalence checkers use an inductive approach to prove equivalence of classes. Thus, we do consider an infinite execution of methods of the given class, while the path based approach would require more effort the more methods are executed.

Other approaches at ESL-to-ESL equivalence checking [53, 36] focus on fine changes to a program and analyze the effects of that change.

Shashidhar et al. [53] decide equivalence when for-loops are restructured or the data flow is changed by introducing or eliminating temporary variables or changing operations by using algebraic properties. The verification is done by generating an *Array Data Dependence Graph* (ADDG) and verifying that the ADDGs are equivalent. The check is done by modifying the ADDGs by using some algebraic transformation. Afterwards, a depth search is done for each output node to guarantee that the some operations are executed on both ADDGs. The programs that can be verified by this approach need to fulfill certain restrictions, e.g., every memory location may only be written once.

Another approach to fine-grained changes [36] detects textual differences between two C programs. Symbolic simulation and validity checking techniques are used to show equivalence of the differences. If this is not successful, the number of statements to be verified is incrementally increased by using the dependency graphs of the programs.

As [53, 36] consider very similar programs with small differences, they consider a different scenario than our equivalence checkers. Our checkers provides a higher level of abstraction which allows two equivalent methods with very different implementation as long as the output is equivalent.

For ESL-to-RTL equivalence checking, several solutions were suggested in academia.

Bounded Model Checking [9] was used to show equivalence of a C program and a Verilog design without focusing on timing. This is realized by extracting a transition relation from both implementations and unwinding them. A SAT solver is used to check if inconsistent, i.e., non equivalent, behavior is possible and returns an according counterexample in that case. Otherwise, the transition relations are further unwound until either the number of unwindings is sufficient to prove equivalence or time or memory bounds are reached. The unrolling does lead to high effort when long paths are considered. Due to our inductive approach, our equivalence checkers can handle long paths with lower effort, especially when a good candidate invariant is given.

A cycle-accurate data-flow graph [29] that combines an RTL and an ESL description into a miter was suggested for equivalence checking. To create the miter, both descriptions are turned into *Data Flow Graphs* (DFG) and the resulting DFGs are combined. Functional equivalence checking can be used on the miter utilizing reachability analysis or induction to detect equivalent variables in the two descriptions. Instead of checking for reachability, Koelbl et al. use k-induction due to its better scalability. In addition, they introduce a constraint DFG, that describes input constraints, output don't-cares, and register mappings. While this allows the incorporation of external knowledge, the provided information needs to be correct. In comparison, the provided

candidate invariant for our equivalence checkers does not need to be a correct invariant and will be adjusted if it is not.

In [29] a miter is generated from the RTL and the ESL descriptions. This approach aims to check equivalence for descriptions that are very different and only share few internal similarities. This difference increases the difficulty of the check and is handled by partitioning all possible execution traces of the miter. Then, a check is done for each partition of traces. This decreases the difficulty of each individual check enough to decrease the overall effort of the equivalence check.

Moreover, Leung et al. [32] propose a translation validation technique for C to Verilog that verifies the equivalence between a C implementation and its Verilog counterpart that has been generated by a High-Level Synthesis tool. They find a bisimulation relation by checking the Verilog code for potential state changes. To speed up the verification process, likely invariants are detected by using Daikon [16]. While Leung et al. formally check whether the generated likely invariants are correct invariants, they cannot correct or adjust them, if the likely invariants are no invariants after all like our equivalence checkers do. In a worst case, the likely invariants are not enough to prove post-conditions and the algorithm will fail on equivalent models.

A semi-formal approach for equivalence checking between circuits at lower abstraction levels is provided in [50]. They compare a circuit model described in Simulink with the corresponding circuit model in SPICE, which contains more details. Simulation data is collected through a number of simulation runs and compared. The difference between the outputs of both models is used to optimize the parameters like voltage of the models until the difference is below a given threshold, as the analog signals of the models usually are not exactly the same.

Gao et al. [21] provide an approach to check if loops in an ESL implementation are executed equivalently in an RTL environment with pipelining. Their approach uses induction and symbolic simulation to handle loops with a large number of iterations. In addition, they check loops for resource conflicts, which easily show that loops are not equivalent with and without pipelining.

As an optimization, equivalence-point detection [17, 3, 18, 62] has been proposed. Equivalence-points are used to separate the execution trace of the two checked models into smaller intervals. If the models are in equivalent states and execute an interval, both models reach an equivalence-point. At this equivalence point, the models need to be in equivalent states if the models are equivalent. These intervals are then used to partition the verification process as only the parts between two equivalence-points need to be checked each run. If all parts are verified to be equivalent, the models can only reach equivalent states after each interval of the execution trace and behave equivalently during the interval. Thus, the models are proven to be equivalent. Otherwise, a single non-equivalent part suffices to prove that the systems are not equivalent.

Instead of using equivalence points, [41] uses trace partitioning to split all possible traces into multiple subsets. This aims to prove equivalence between ESL and RTL for very different implementations. In these cases, equivalence points are rare and do not help to speed up the process. On the other hand, trace partitioning allows to split the complex proof of equivalence into smaller sub-proofs that are solved individually. If all sub-proofs are executed successfully, the models are equivalent. Otherwise, a counterexample to a single sub-proof suffices to disprove equivalence.

2.4 Fault Models

There is a number of causes that lead to faults within a finished system. During fabrication, manufacturing defects can make unwanted changes to an integrated circuit and change its behavior. Some wires may not be connected or an additional and unwanted connection may occur. Another effect during fabrication is process variation [38], which especially affects smaller transistors. It describes naturally occurring variation in the attributes of transistors may cause faults.

Aging affects the system later on during its life cycle [55]. As transistors degrade over time, faults can occur. The performance of transistors decreases in time, which can lead to increased delays, or existing connection may break.

Another source for faults is radiation [25]. Ionized radiation can give a charge to parts of the system. This charge can, depending on its energy, temporarily change the behavior of that part or even cause permanent faults up to complete destruction. These effects are especially relevant in outer space or aviation where cosmic radiation has a significantly higher effect than on ground level as the atmosphere of Earth absorbs most radiation, but with the decreasing size of transistors the charge that can cause a fault decreases and the effects of cosmic radiation become more relevant on ground level. Nuclear reactors or other sources for ionized radiation can cause similar effects. Thus, when a system is meant to be used in a radiation rich environment, it needs to handle the corresponding faults.

Finally, system can be affected by physical harm. Extreme heat or cold can cause harm to the system, friction or other external forces can damage the system, and so on. These effects can cause faults by breaking parts of the system or cause interactions by unfortunately modifying the system and, for example, causing connections that should not exist.

When developing a system, the requirements define which kind of faults need to be handled by the system. The planned level of robustness depends on different factors. It needs to be considered in which environment the system will be used and how critical the system itself is. While it is unfortunate when a gaming console breaks due to errors, no lives are put at risk. The effect of errors on safety critical systems like cars and planes are far more severe and can in a worst case scenario cause the loss of life.

One way to check the robustness of a system is applying the causes for faults to a prototype of the system. For example, satellites are tested thoroughly before they are sent to space [4]. Shakers ensure that the satellite can handle the vibration during launch, drop tests show that the satellite can survive brief shocks, proton beams are used to analyze the effects of radiation, and so on.

While these approaches provide a very realistic environment for the system, they require a working prototype. Thus, they can only be applied at late stages of the development. Furthermore, these tests are time-consuming and costly. Finally, these tests will not produce all possible scenarios that can effect the system and are as such not extensive.

To analyze the effects of faults earlier within the development cycle and enable easier and automated testing or verification, fault models are used. Fault models can describe the effects of faults on the system at different abstraction levels without requiring to considering the the exact causes of the fault. Thus, when a system is deemed robust against the considered fault models, it is assumed to be robust against the corresponding real faults as well.

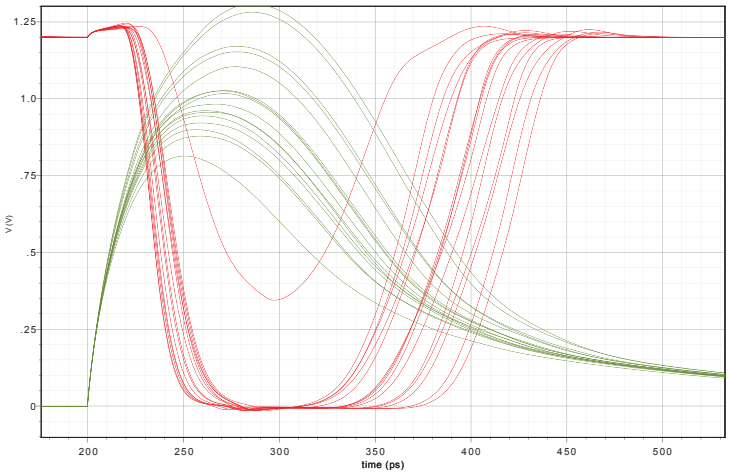


Figure 2.3: SETs affecting a signal

Fault models describe permanent or transient faults. Once a permanent fault affects the system, its effect remains indefinitely. A prominent example for permanent faults are *stuck-at faults* [35]. A stuck-at fault assumes that a signal within the circuit is fixed at a specific value and cannot be affected by the corresponding inputs. The output of an `or`-gate that is stuck-at-0 would always produce a 0 even if inputs of the `or`-gate are set to 1 and the correct output of the gate would be 1 as well. This fault model is mostly used to describe manufacturing defects. Test sets are applied to each produced system to show that the system is not affected by stuck-at faults and is usable. Other examples for permanent faults are *bridging faults* [14], that describe additional wires within the system that connect parts that should not be connected, or *delay faults* [55], which describe delays of gates within the system that could change the behavior of the system.

On the other hand, transient faults affect the system only for a limited time. While the fault itself only shortly affects the system, the effects of the fault can become permanent, e.g., a signal that is changed for a short duration can still lead to an error that permanently affects the system by changing the systems state. Most transient faults are caused by radiation. The faults that can arise due to radiation are categorized as different *Single Event Effects* (SEE) [47]. An SEE models the effects of a single energetic particle. The particle releases its charge within the system and causes a voltage glitch within the system. Depending on the location, this glitch can have different effects on the system. While most SEEs are transient, it is also possible that an SEE is a permanent fault.

Single Event Upsets (SEU) [43, 56] describe the effect when an SEE affects the registers of the system and changes the current state. As the system is in a faulty state afterwards, its behavior may be different than it should be.

An *Single Event Transient* (SET) [20] on the other hand affects the combinatorical part of the system and changes the output of a gate within the circuit for a short time. An SET can traverse through the circuit. If it affects a primary output, it causes an error, and if it affects a register, the system will assume a faulty state which can lead to errors later on. Some examples for SETs with different particle energies are shown in Figure 2.3. The figure shows how the nominal value of the affected signal is changed for a short duration. The green lines present changes from 0 to 1 while the red lines show a change from 1 to 0. The value does not change instantly as the increasing of voltage takes some time and especially the return to 0 after an SET changed the signal value to 1 requires more time as the additional energy is released again. If the values within the circuit are considered as binary, the interpretation during this time is uncertain.

Single Event Latchups (SEL) can latch systems that use thyristors into a high current state. This arises due to the SEE affecting the inner-transistor junctions. If the charge is too high, an SEL can harm the system irreversible. Otherwise, the effects of the SEL can be removed by resetting the device.

While SEUs, SETs, and SELs are the most prominent SEEs, there are others. A *Single Event Snapback* (SES) is similar to an SEL but is caused by an SEE in the drain junction of a N-channel *Metal–Oxide–Semiconductor* (MOS) transistor and results in a high current state. A *Single Event Burnout* (SEB) describes that a device draws high current and burns out. *Single Event Gate Ruptures* (SEGR) destroys a gate in a power *Metal–Oxide–Semiconductor Field-Effect Transistor* (MOSFET). Finally, a *Single Event Functional Interrupt* (SEFI) describes a corrupted control path due to an upset.

2.5 Hardening Methods to Provide Robustness

As the system needs to be robust against certain faults, the system needs to be hardened accordingly. Depending on the considered faults, a developer can choose from multiple techniques to harden the system. Unfortunately, hardening always causes overhead in one way or another.

The usual approach to harden a system is redundancy. If some redundant element is affected by a fault, this fault is meant to be corrected or detected due to the additional elements. If fault detection is used, the system usually reverts back into a safe state after a fault has been detected and redoes the previously faulty operation. Redundancy can be split into different categories: hardware, timing, information, and software.

Hardware redundancy describes that more hardware elements are used to provide redundancy. These multiple elements all provide the same functionality and therefore can detect or even correct faults. To handle permanent faults, some elements may be offline and are only activated when needed due to other elements failing. A common hardening from this category is *Triple Modular Redundancy* (TMR) [34], which means that a system is triplicated and a voter decides the primary outputs. If one copy of the system is faulty, the other copies correct that fault.

If timing redundancy is used, the redundancy is achieved by providing the system additional time to execute operations. For example, a primary output could be computed multiple times with some delay between each computation.

Afterwards, a voter decides over the final value. *Timed TMR* (TTMR) [42] describes this process for three differently delayed output times. As timing redundancy usually uses the same hardware to execute the values over time, it is meant to handle transient faults. Permanent faults cannot be handled by pure timing redundancy.

Fault-detecting or -correcting codes [27] like Hamming-Code are examples for information redundancy. Additional data is stored and can be used to either detect or even correct a fault within the date.

The idea of software redundancy [48] is similar to hardware redundancy but is applied on a different level. Different teams develop the same subsystem in respect to the same requirements but use different approaches. Like hardware redundancy, the multiple versions of the subsystem can be used to correct or detect faults. In addition, if all versions are affected by the same fault, each copy handles the fault differently and it is possible that the fault can still be handled. And if faults remain within the implementation of the versions, these faults can be corrected as long as the implementation faults within the different versions do not overlap.

2.6 Analyzing Robustness

After hardening has been applied to the system, it needs to be verified that the hardening protects the system against the considered faults and thus provides robustness. Robustness can be checked by using testing or formal verification.

In case of using testing [30], a certain set of test cases is run on the system. This is done similarly to regular testing approaches. In addition, a fault is injected in each test case, that follows the restrictions from the fault model. The fault can be random or given with the test case. If the output of the system remains correct and is not changed by the injected fault for all test cases, the system is assumed to be robust. However, as extensive testing is only feasible for very small systems, it usually cannot guarantee robustness.

A common method to analyze the robustness of a system is Monte Carlo Simulation [39]. In this approach, the developer defines a set of possible input values, including faults. From this set, a number of evenly distributed test cases is randomly generated and executed on the system. While this approach cannot prove the absence of errors, it can usually estimate the probability of an error very close to the real probability.

Formal verification can prove the absence of faults by modeling all possible executions of the system or a subsystem with all possible input values and faults corresponding to the considered fault model. The downside of this approach is that models can become very complex if the detailed behavior of bigger systems is considered. If the models are too complex, a solver cannot handle them within a feasible time. The complexity can be reduced by focussing on certain parts of the system only or by abstracting some details. If details are abstracted, it is important that the results gained from the abstract model are still applicable to the real system.

Different work about the formal verification of transient faults exists and focuses on different aspects. Several techniques focus only on logical masking [19, 24, 33, 52] which leads to quick decisions as the abstraction levels like register or gate level can easily be decided but exclude finer details about the behavior

of the circuit.

Frehse et al. [19] use an approach based upon bounded model checking. For each gate within the circuit they compute whether the gate is robust, non-robust, or dangerous. If the gate is robust, a fault within that gate is corrected and the primary outputs are not affected and after a number of clock cycles the system reaches the same state as it would without the fault. A gate is also categorized as robust if a fault signal is set within the circuit if a fault affects that gate. Faults in non-robust gates affect the primary outputs after a number of clock cycles without triggering the fault signal. If a fault in a gate does not affect the primary outputs but can permanently cause a faulty state of the system, it is deemed dangerous unless the fault signal is set.

Leveugle [33] uses controlled generation to produce a mutant of a circuit given in VHDL. The mutant inserts additional signals that can be used to insert faults. In a next step, formal property checking is used to check if the properties for the original circuit hold on the mutant as well. If the properties hold, the circuit executes correctly, even under all possible injectable faults. Otherwise, the property check generates a counterexample that can be used to guide the further hardening of the circuit.

Similarly to [33], Seshia et al. [52] check if the properties of the original circuit hold under faults as well. They generate one formal model for each latch that can be affected by an SEU. The formal model describes the behavior of the circuit under an SEU in the corresponding latch. If an SEU in a latch leads to an error, the latch needs to be hardened. In the end, the circuit is synthesized with all problematic latches hardened. If the estimations of power, performance, and other circuit parameters are sufficient, the process is finished. If the parameters are not sufficient, a designer needs to adjust the circuit manually and execute the verification again.

In [24], a probabilistic model is suggested, that can be used to compute the probability that faults cause an error. In this model, every gate has a probability to be affected by a fault and each primary input has a certain probability to be. Different fault models like SETs or stuck-at faults can be described by using different probability functions for the output of a gate. Using these probabilities of inputs and faults, it is possible to compute the probability that a primary output returns an erroneous value.

Shazli et al. [54] use a SAT-based approach to determine the probability for an error. They consider the system at RTL and construct a SAT instance that is satisfied under an error. They use a solver to determine all solutions for the instance and determine the probability as the number of solutions divided by the number of possible assignments.

Other work emphasizes the effects of timing masking under delay faults [51, 11] and does not only consider logical masking but timing masking as well.

Sauer et al. [51] use waveforms to describe the change of each signal within a circuit over time. They provide a tool that generates a test case that can observe a given *Small Delay Fault* (SDF). An SDF describes that the delay of a specific gate is increased by a certain duration. The tool generates a SAT instance that requires that a fault is observable, i.e., turns into an error. Then, a SAT solver is used to find a solution, which corresponds to a test case to observe the given SDF. If the SAT instance is not satisfiable, the SDF is not observable.

The opposite scenario of [51] is described in [11]. They start from a failed test case and detect a minimal set of SDFs that can cause the described erroneous

Approach	Masking effects			result
	logical	timing	electrical	
[19]	✓			decide robustness
[33]	✓			decide robustness
[52]	✓			hardened circuit
[24]	✓			probability of error
[54]	✓			probability of error
[51]	✓	✓		create test case
[11]	✓	✓		minimal set of SDFs
[45]			✓	probability of error
[23]	✓		✓	probability of error
[37]	✓	✓	✓	probability of error
our algorithms	✓	✓	✓	decide robustness

Table 2.1: Approaches to analyze robustness

output. In this scenario, the primary inputs and outputs of the circuit are set according to the test case. In addition to the original circuit model, an additional signal is added for each gate, that is set if the gate is affected by an SDF. The number of these signals that can be set is limited by an upper threshold. A SAT solver detects a set of SDFs that can cause the described error or proves that a number of SDFs given by the threshold cannot cause the error. To detect a minimal set of SDFs, the threshold is initially set to 1 and increased whenever no solution is detected.

So far, the related work analyzed logical and timing masking, but electrical masking was not considered. The following contributions [45, 23, 37] do consider these masking effects.

In [45] only electrical masking is considered. They define a model that describes if a transient fault can propagate through a gate. The propagation does only consider the voltage and the duration of a glitch to estimate the probability for the fault to reach the primary outputs. While their approach is not precise, it is significantly faster than spice simulation while providing 90% accuracy compared to spice.

The probability of an error is estimated in [23] by evaluating an SMT formula that describes the propagation of an SET in a combinatorical circuit. They consider logical as well as electrical masking. They do not consider the effects of reconverging signals and avoid the corresponding complexity.

Miskov-Zivanov [37] compute the probability of an error under multiple transient faults within a circuit due to a particle strike. To determine gates that can be affected at the same time, a neighborhood relation is defined. The model considers logical, timing, and electrical masking and describes the effects of the transient fault and its propagation by using Binary Decision Diagrams (BDDs) and Algebraic Decision Diagrams (ADDs).

Table 2.1 summarizes the presented related work and compares them against our algorithms for robustness checking. Beside the differences shown in the table, our algorithms are currently the only formal approaches to consider variability when analyzing the robustness of a circuit against SETs.

Chapter 3

System Level Equivalence Checking

ESL design methodologies focus on the description of the functionality of an entire system on a high level of abstraction. In contrast to traditional RTL descriptions, an ESL description captures the behavior of a system while neglecting low-level details like hardware/software partitioning, timing, or power consumption. This allows designers to focus on behavioral characteristics of the system and enables functional verification and validation in early design phases. ESL descriptions are often formulated in abstract programming languages like SystemC or general-purpose programming languages like Java or C++.

In this chapter, we introduce, NSMC and EASY[1], two algorithms for functional equivalence checking on the ESL. Both algorithms take as input two ESL descriptions to be checked for functional equivalence, corresponding mappings between the initial states and operations of the two descriptions, and optionally a candidate invariant. In essence, the algorithms systematically learn and improve an invariant that characterizes the reachable states of a miter of the two ESL descriptions, until either an inductive correctness proof succeeds or a counterexample has been found that disproves functional equivalence. While NSMC advances the given candidate invariant, EASY uses a *property-directed reachability* (PDR) [7, 13] approach refine the invariant. On termination, the learned invariant serves as a certificate for functional equivalence, whereas a counterexample can be used for debugging the ESL descriptions. The correspondence mappings are necessary to match the two ESL descriptions if they are structurally different. Optionally, a candidate invariant can be provided to approximate the reachable states of the miter of the two ESL descriptions; underapproximation as well as overapproximations are supported. The candidate invariant is a simple way to incorporate knowledge a priori known by the designer into the verification process to speed up reasoning. If the provided candidate invariant indeed is inductive, the algorithm terminates quickly as equivalence can be shown easily. Otherwise, in an attempt to prove functional equivalence, the algorithms iteratively refines the candidate invariant utilizing counterexamples when functional equivalence checking fails. A counterexample is either spurious, i.e., unreachable from the initial states, then those states can safely be excluded

[1] pronounced as the two letters E.C.

from the candidate invariant, or a counterexample is real, such that a mismatch of the behavior of the two ESL descriptions has been revealed.

Compared to the previous work described in Chapter 2.3, our algorithms are the only ones that exploit designer knowledge for a formal equivalence check. Closest to our approach, [32] uses potential invariants to detect cutpoints in the execution traces and speeding up the decision. Moreover, the formal approaches used by the previous work are either bounded model checking or k-induction to prove equivalence while EASY uses the concepts of PDR.

Contribution. This chapter makes the following contributions.

1. We describe a light-weight design and verification methodology for embedded systems on the ESL. The behavior of the system is described on a high abstraction level utilizing C++ as flexible modeling language. A system is described as a C++ class — the member variables describe the system's state, whereas the methods describe operations that manipulate the state. The C++ code serves as an executable, functional specification of an embedded system neglecting low-level design details.

2. We present two algorithms, NSMC and EASY, state-of-the-art algorithms to prove or disprove functional equivalence of ESL descriptions that follows the described design methodology and especially allows to incorporate designer knowledge to speed up the reasoning process. On termination, the algorithms produce a certificate in terms of an inductive invariant if the two ESL descriptions are functionally equivalent or a counterexample if functional equivalence was disproved.

3. We provide an implementation of both algorithms that instruments the given C++ classes with a simple assertion checking scheme and uses CBMC [8] as model checker.

4. In our experiments, we compare NSMC and EASY against each other and show the advantages of each algorithm. The experiments show that EASY usually decides equivalence faster than NSMC and can decide equivalence within a feasible time for more cases than NSMC. We also show that a good hypothesis enables a decision on more complex examples.

The remainder of the chapter is structured as follows: first, we present preliminaries and our deduced models in Section 3.1. Then, the core algorithms are proposed. Section 3.2 shows the initial algorithm NSMC and Section 3.3 shows the PDR-based algorithm EASY. Lastly, we present our experiments with three examples in Section 3.4. Section 3.5 concludes.

3.1 Preliminaries and Used Models

In this section, we introduce a variant of *Mealy transducers* [63] to model hardware modules and characterize the functional equivalence of two Mealy transducers based on their input/output behavior. Furthermore, we will describe how the hardware modules on ESL are modeled and what data structures we use to support the equivalence check. In Section 3.1.1, we will show how the hardware modules given as C++ classes are described as Mealy Transducers.

Section 3.1.2 will introduce lockstep machines, which are used to combine two Mealy transducers into a single new Mealy transducer, and in Section 3.1.3 we will show how logical formulas are used to reason over subsets of states.

Definition 1. A *Mealy transducer* $M = (S, S_0, X, Y, \phi, \psi)$ is a tuple, where S is a finite non-empty set of states, $S_0 \subseteq S$ is the finite subset of initial states, X is the input alphabet, Y is the output alphabet, $\phi : S \times X \to S$ is the transition function, and $\psi : S \times X \to Y$ is the output function.

For an input $x = x_0 x_1 \dots x_n \in X^*$, we say that $y = y_0 y_1 \dots y_n \in Y^*$ is an output of M if there exists a state sequence $s_0 s_1 \dots s_{n+1} \in S^*$, such that $\forall i \in \mathbb{N}_0, i \leq n : \phi(s_i, x_i) = s_{i+1} \wedge \psi(s_i, x_i) = y_i$, i.e.,

$$s_0 \to^{x_0/y_0} s_1 \to^{x_1/y_1} \dots \to^{x_n/y_n} s_{n+1},$$

where $s_0 \in S_0$ and $s_i \in S$ for $0 \leq i \leq n+1$. We write $y \in M(x)$, where $M(x)$ is the set of all outputs produced by M for input word x.

In contrast to the standard definition of [63], our definition of Mealy transducers does not define any accepting or final states, but assumes that all states are accepting.

Definition 2. Two Mealy transducers M and M' are *functionally equivalent* iff they produce the same output words for all input words, i.e., $M(x) = M'(x)$ for all $x \in X^*$.

3.1.1 Modeling Hardware Modules

We model hardware modules on system level as C++ classes. We utilize the Mealy transducers from [63] to describe the behavior of a C++ class. The member variables of a class define the state of the hardware module, whereas the public methods of the class with its arguments define terminating operations that can be executed to change the state and describe the inputs of the according Mealy transducer. In our model, each possible argument of a method defines a different input to the Mealy transducer. However, in our checks, we will consider the arguments nondeterministically. Note that we are not interested in modeling the interior behavior of a method but consider only the states when entering and leaving the method. Thus a method is seen as one atomic and terminating operation.

Consider the counter example from Section 2.1. In Section 2.1 we only considered a single model at ESL. This time, let us assume an iterative development process with multiple ESL models. Figure 3.1 shows two implementations of the counter modeled in C++: Figure 3.1a declares the interface of the counter as a class. The class has one member variable `counter` which stores the actual state of the counter and is initialized to 0 and one method `countUp` to increase its value. The two implementations in Figure 3.1b and Figure 3.1c use the standard binary encoding of unsigned integers to represent the counter value but differ in the approach to reset the value when the counter increases beyond its 2-bit range. The first implementation uses a modulo operator to stay within the counting range when counting up, whereas the latter uses a conditional statement to reset the counter to 0 when the value 3 is increased.

```
class Counter {
public:
  uint8_t countUp();
private:
  uint8_t count = 0u;
};
```

(a) Interface

```
uint8_t Counter::countUp() {
  count = (count+1u) % 4u;
  return count;
}
```

(b) Implementation #1

```
uint8_t Counter::countUp() {
  if ( count == 3u )
    count = 0u;
  else
    count = count+1u;
  return count;
}
```

(c) Implementation #2

Figure 3.1: Interface and different implementations of a modulo-4 counter

Example 1. Suppose that $\mathsf{Int}_k = \{0, 1, \ldots, k-1\}$. The Mealy transducer $M_{\mathsf{mod}} = (\mathsf{Int}_{256}, \{0\}, \{\texttt{countUp}\}, \mathsf{Int}_4, \phi_{\mathsf{mod}}, \psi_{\mathsf{mod}})$ and the Mealy transducer $M_{\mathsf{if}} = (\mathsf{Int}_{256}, \{0\}, \{\texttt{countUp}\}, \mathsf{Int}_{256} \setminus \{4\}, \phi_{\mathsf{if}}, \psi_{\mathsf{if}})$ model the input/output behavior of the two implementations in Figure 3.1b and Figure 3.1c, respectively, where for $i \in \mathsf{Int}_{256}$

$$\psi_{\mathsf{mod}}(\texttt{countUp}, i) = (i+1)\%4 \text{ and}$$
$$\psi_{\mathsf{if}}(\texttt{countUp}, i) = \mathit{count}_{\mathsf{if}}(i) \text{ with}$$
$$\mathit{count}_{\mathsf{if}}(i) = \begin{cases} 0, & i = 3 \\ i+1, & \text{else.} \end{cases}$$

Finally, the next-state function and the output function are equal, i.e., $\psi_{\mathsf{mod}} = \phi_{\mathsf{mod}}$ and $\psi_{\mathsf{if}} = \phi_{\mathsf{if}}$.

The state spaces and transition functions of the two counter implementations are visualized in Figure 3.2 (M_{mod} on the top and M_{if} on the bottom). Each node in the figure corresponds to a possible state and each edge from u to v indicates that state v is reached when the method `countUp` is executed in state u. The initial nodes are marked with an additional incoming edge. The output produced in each state is identical with the counter value in the reached state. Both implementations behave equivalently within the states reachable from their initial states and thus M_{mod} and M_{if} are intuitively functionally equivalent according to Definition 2.

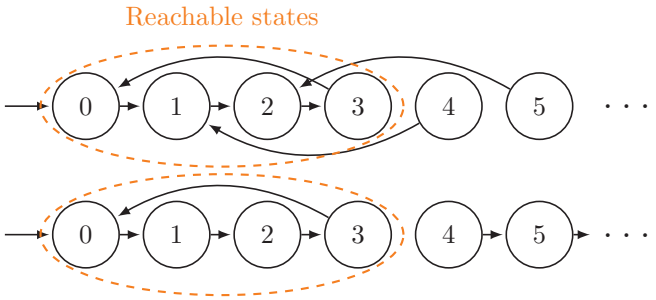


Figure 3.2: Visualized state spaces and transition functions of M_{mod} (top) and M_{if} (bottom).

3.1.2 Lockstep Machine

We use a lockstep machine to describe the parallel execution of methods on the two considered hardware modules as we want to check if both modules behave in the same way when methods that should be equivalent are called.

Definition 3. Consider the Mealy transducers $M_1 = (S_1, S_{0_1}, X_1, Y_1, \phi_1, \psi_1)$ and $M_2 = (S_2, S_{0_2}, X_2, Y_2, \phi_2, \psi_2)$. A *lockstep machine* of M_1 and M_2 is a tuple $M_* = (S_* = S_1 \times S_2, S_{0_*} = S_{0_1} \times S_{0_2}, X_* = X_1 \times X_2, Y_* = Y_1 \times Y_2, \phi_*, \psi_*)$, where

$$\phi_* : S_* \times X_* \to S_*, \left(\begin{pmatrix} s' \\ s'' \end{pmatrix}, \begin{pmatrix} x' \\ x'' \end{pmatrix}\right) \mapsto \begin{pmatrix} \phi_1(s', x') \\ \phi_2(s'', x'') \end{pmatrix} \text{ and}$$
$$\psi_* : S_* \times X_* \to Y_*, \left(\begin{pmatrix} s' \\ s'' \end{pmatrix}, \begin{pmatrix} x' \\ x'' \end{pmatrix}\right) \mapsto \begin{pmatrix} \psi_1(s', x') \\ \psi_2(s'', x'') \end{pmatrix}.$$

Definition 4. Suppose that $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$ is a lockstep machine of the Mealy transducers M_1 and M_2. Then the pair $(\delta \subseteq S_{0_*}, \Delta \subseteq X_*)$ is called a *correspondence mapping*.

The set δ describes initial states that are meant to be equivalent and is called a *state mapping* while Δ is a *method mapping* that describes which methods of M_1 and M_2 are meant to be equivalent.

A state $s = (s', s'') \in S_*$ is called *safe* under Δ iff $\psi_1(s', x') = \psi_2(s'', x'')$ for all $(x', x'') \in \Delta$ and *unsafe* otherwise.

Moreover, M_* is called *equivalent* under (δ, Δ) iff for all finite sequences $x = x_1 x_2 \dots x_n \in \Delta^n$ of methods and all initial states $s_0 \in \delta$, M_* the sequence $s_0 s_1 \dots s_n \in S_*^{n+1}$ reaches a safe state s_n, where $s_i = \phi_*(s_{i-1}, x_i)$ for $1 \leq i \leq n$.

Lemma 1. M_* is equivalent under (δ, Δ) with $\Delta = \{(x', x') | x' \in X_1 \cap X_2\}$ iff M_1 and M_2 are functionally equivalent for every $(s'_0, s''_0) \in \delta$.

Proof. 1. Let $\Delta = \{(x', x') | x' \in X_1 \cap X_2\}$ and let M_* be equivalent under (δ, Δ). Consider any initial state $s_0 = (s'_0, s''_0) \in \delta$ and any finite sequence $x = x_1 x_2 \dots x_n \in (X_1 \cap X_2)^*$. Let $M_1(s'_0) = y'_1 y'_2 \dots y'_n$ and $M_2(s''_0) = y''_1 y''_2 \dots y''_n$. To prove that M_1 and M_2 are functionally equivalent for s_0, we need to show that

$$\forall i \in \mathbb{N}_0, i \leq n : y'_i = y''_i$$

After the finite sequence $(x'_1, x'_1)(x'_2, x'_2) \dots (x'_{i-1}, x'_{i-1}) \in \Delta^{i-1}$ of method calls for any $i \in \mathbb{N}$ with $i \leq n$, M_* reaches a safe state (s', s'') from s_0 since M_* is equivalent under (δ, Δ). Therefore,

$$\exists \tilde{y} \in Y_1 \cap Y_2 : \psi((s', s''), (x'_i, x'_i)) = (\tilde{y}, \tilde{y}) \\ \Rightarrow y'_i = y''_i = \tilde{y}$$

Since the output sequences of M_1 and M_2 are equal, they are functionally equivalent for all $s_0 \in \delta$ when M_* is equivalent under (δ, Δ).

2. Let the Mealy transducers M_1 and M_2 be functionally equivalent for every $(s'_0, s''_0) \in \delta$. Consider any finite sequence $x = x_1 x_2 \dots x_n = (x'_1, x'_1)(x'_2 x'_2) \dots (x'_n x'_n) \in \Delta^n$ with $x'_i \in X_1 \cap X_2$ for $1 \leq i \leq n$ and any initial state $(s'_0, s''_0) \in \delta$.

   Let the state of M_1 after the sequence $x' = x'_1 x'_2 \dots x'_n$ in s'_0 be s' and the state of M_2 after the method sequence x' in s''_0 be s''. Then M_* will be in the state (s', s'') after the method sequence x due to its construction.

   We need to show that (s', s'') is safe. Consider any $(\tilde{x}, \tilde{x}) \in \Delta$ and the path $x \cdot (\tilde{x}, \tilde{x}) = (x'_1, x'_1)(x'_2, x'_2) \dots (x'_n, x'_n)(\tilde{x}, \tilde{x})$. The sequence $x' \cdot \tilde{x}$ will lead to the same output in M_1 and M_2 due to them being functionally equivalent, i.e., $M_1(s'_0, x' \cdot \tilde{x}) = M_2(s''_0, x' \cdot \tilde{x}) = y'_1 y'_2 \dots y'_n \tilde{y}$. Since the sequence x leads M_* from (s'_0, s''_0) to (s', s''), $\psi_*((s', s''), (\tilde{x}, \tilde{x})) = (\tilde{y}, \tilde{y})$.

   Since $\tilde{y} = \tilde{y}$, the state (s', s'') is safe and therefore M_* is equivalent under (δ, Δ) when M_1 and M_2 are functionally equivalent for every $(s'_0, s''_0) \in \delta$. □

In addition, EASY enables us to define correspondences between different numbers of methods, e.g., the call of one method in an abstract model corresponds to the call of multiple methods in the detailed model. All future checks will work equivalently in these cases even though we will assume a one-to-one mapping in our explanations.

3.1.3 Candidate Invariant and Learned Clauses

The states in our lockstep machines describe complete assignments of all member variables of the currently checked models. The evaluation of a formula f under the assignment that is described by a state s is written as $f(s)$. We say that a state s fulfills a formula f if $f(s) = \text{true}$.

A logical formula f defines a set S_f of states. The set S_f contains exactly all states that fulfill f.

We use a model checker for individual steps during our equivalence check. A call to the model checker uses a *pre-* and a *post-hypothesis*. These hypotheses are formulas over the variables of the checked models and are usually the candidate invariant or a subformula of the candidate. A modelcheck with such a pair of hypotheses checks if all states that fulfill the pre-hypothesis are safe and reach a state that fulfills the post-hypothesis after the execution of any method $m \in \Delta$. When doing a modelcheck, we verify the existence of a counterexample for each method separately but consider all possible arguments of that method nondeterministically. A call to the model checker is given as

$\mathsf{Check}(h_{\text{pre}}, h_{\text{post}}, M_1, M_2, \Delta)$. The call creates a miter for each pair of methods in Δ our underlying model checker and returns a *counterexample* or $\perp$ if no counterexample exists.

Definition 5. A *counterexample* is a triple $cex = (o, r, m)$ and depends on a lockstep machine $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$, a pre- and a post-hypothesis h_{pre} and h_{post}, and a method mapping Δ. A counterexample describes an originating state $o \in S_*$ that fulfills h_{pre}. When the method $m \in \Delta$ is called in o, the state $r \in S_*$ is reached, i.e., $\phi_*(o, m) = r$. In the counterexample the method returns non-equivalent output, i.e., $\exists(y', y'') \in Y_* : y' \neq y'' \wedge \psi_*(s, m) = (y', y'')$, or the reached state does not fulfill the post-hypothesis, i.e., the assignment of variables given by r does not satisfy h_{post}.

As we cannot directly know if a counterexample describes a non-safe state or a reachable state that does not fulfill the post-hypothesis, we can use the post-hypothesis "true" to specifically check for non-safe states without utilizing another interface.

When using EASY, we do not consider a single candidate invariant at a time, but utilize multiple sets of states that overapproximate states reachable in certain numbers of steps similar to PDR.

Definition 6. Consider a lockstep machine $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$ and an according correspondance mapping (δ, Δ). A *state vector* with highest index N is a sequence of sets of states $\overrightarrow{S} = S_0 S_1 \dots S_N \in S_*^{N+1}$. Each set S_i is an overapproximation of all states that are reachable within i or less steps from a state of δ and is called the i-th *frame*. The set S_0 is a special case and contains all initial states from δ and as such $S_0 = \delta$. Different from the other sets, S_0 is an exact representation of all states that are reachable within 0 steps and will not be changed during the algorithm while the other sets will be refined if needed to prove or disprove equivalence.

When running the algorithm, the sequence $S_0 S_1 \dots S_N$ needs to fulfill two properties:

$$\forall i \in \{0, 1, \dots, N-1\} \forall s \in S_i : s \text{ is safe} \tag{3.1}$$

$$\forall i \in \{0, 1, \dots, N-1\} \forall s \in S_i \forall m \in \Delta : \phi_*(s, m) \in S_{i+1} \tag{3.2}$$

Property 3.1 states that all sets of states from the state vector except for S_N only contain safe states. As such, we must ensure that all S_N also only contains safe states before increasing N.

The second property 3.2 describes, that for all $i \in \{0, 1, \dots, N-1\}$, when any method $m \in \Delta$ is called in a state $s \in S_i$, the state that is reached after the execution of m must be contained in S_{i+1}. As $S_0 = \delta$, property 3.2 ensures that each set S_i is an overapproximation of states that are reachable within i steps.

During the algorithm we will start with $N = 1$ and increase N whenever S_N only contains safe states. Initially S_1 will contain all states that that fulfill the candidate invariant.

Definition 7. A *clause vector* $\overrightarrow{F} = F_0 F_1 \dots F_N$ with highest index N is a vector, where each F_i is a set of *clauses*, i.e., logic formulas, and defines a state vector $\overrightarrow{S}$ under the lockstep machine $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$.

We define a combined formula $F_i\uparrow$ as a conjunction over all clauses of $F_i, F_{i+1}, \dots, F_N$, i.e.,

$$F_i\uparrow \;=\; \bigwedge_{j\in\{i,i+1,\dots,N\}} \bigwedge_{c\in F_j} c.$$

The state vector $\overrightarrow{S} = S_0 S_1 \dots S_N$ that corresponds to $\overrightarrow{F}$ is defined such that for every $i \in \{0, 1, \dots, N\}$

$$S_i = \{s \in S_* | F_i\uparrow (s) = \text{true}\}$$

For better readability, we sometimes use a set of clauses to describe a formula. In this case, the described formula is a conjunction of all clauses within the set. In addition, sets of states and formulas are sometimes used interchangeably. When a formula is used in place of a set of states, the set contains exactly all states that fulfill the formula. In the other case, when a set of states is used to describe a formula, the resulting formula is fulfilled for a state iff that state is within the set.

3.2 Our initial algorithm NSMC

We present an algorithm that does an equivalence check between two high-level models of a hardware system described in C++. The algorithm is sketched in Figure 3.3. The inputs for the algorithm are the two models M_1 and M_2, the candidate invariant c, and a correspondence mapping (δ, Δ). Initially, the candidate invariant is modified to include all initial states. This is required for further steps to properly detect if states of the lockstep automata are reachable. Next, c is modified by excluding unsafe states as well as part of their predecessors from the candidate invariant.

If an initial state $s_0 \in \delta$ was removed during this step, $\neg c \wedge \delta$ is satisfiable and the models M_1 and M_2 are not equivalent under (δ, Δ). Otherwise, we check if the candidate invariant is inductive. If there are no more counterexamples against c being inductive, the models are proven to be equivalent. Otherwise, a counterexample (o, r, m) exists that disproves equivalence. As all unsafe states have already been removed from the candidate invariant, the state r does not fulfill the candidate invariant. A formula *preds* is generated that describes o and its predecessors. If *preds* does not describe any initial states o and all other states in *preds* are not reachable. In this case, $\delta \wedge$ *preds* is unsatisfiable and we can safely remove *preds* from the candidate invariant and check if there are further counterexamples. Otherwise, we know that r is reachable. We check if r is a safe state. If not, we have disproven equivalence. Otherwise, we modify c by adding r to the candidate invariant and check for more counterexamples.

The following sections explain our algorithm in higher detail. Section 3.2.1 presents our top level algorithm. Sections 3.2.2, and 3.2.3 present underlying functions of our algorithm and provide an even higher grade of detail. Finally, in Section 3.2.4, we sketch why NSMC always terminates and decides correctly.

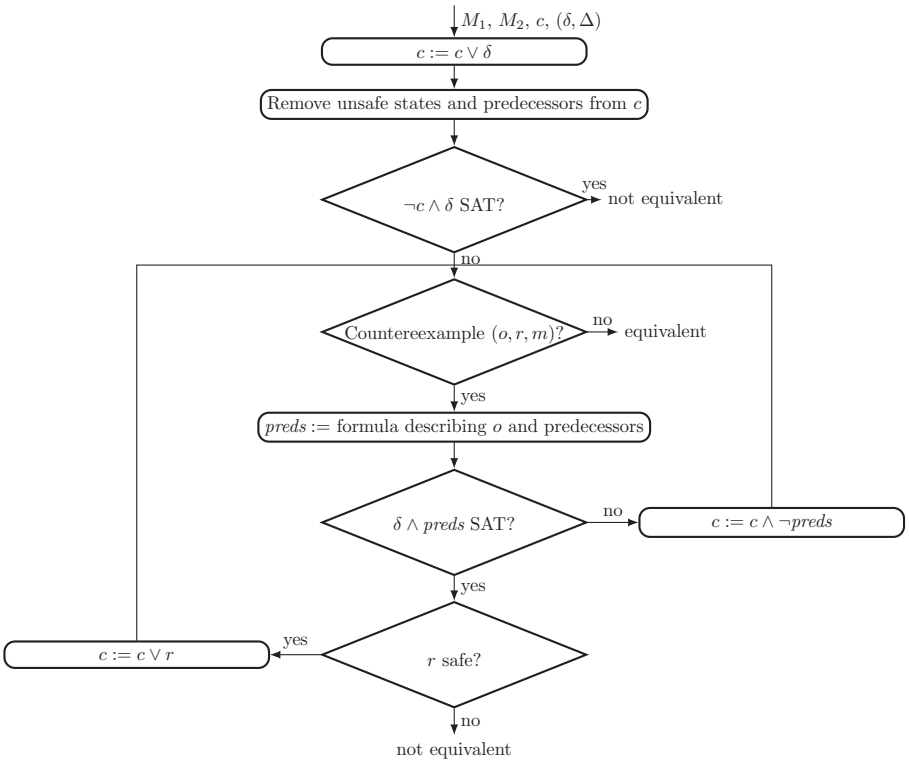


Figure 3.3: Checking equivalence between M_1 and M_2 under (δ, Δ)

Algorithm 1: NSMC

```
   input  : two Mealy transducers M1 and M2 of C++ classes, a
            correspondance mapping (δ, Δ), and a candidate invariant c
   output : An invariant if the models are equivalent and ⊥ otherwise
 1 c := c ∨ δ
 2 c := c ∧ ¬ PRED-STATES(M1, M2, Δ, c, true)
 3 if δ → ¬c then return ⊥
 4 while (cex := Check(c, c, M1, M2, Δ)) ≠ ⊥ do
 5 |   preds := PRED-STATES(M1, M2, Δ, c, ¬cex.r)
 6 |   if δ → preds then
 7 |   |   if Check(cex.r, true, M1, M2, Δ) ≠ ⊥ then
 8 |   |   |   return ⊥
 9 |   |   else
10 |   |   |   reach := cex.r
11 |   |   |   while (cex' := Check(reach, c ∨ reach, M1, M2, Δ)) ≠ ⊥ do
12 |   |   |   |   if Check(cex'.r, true, M1, M2, Δ) ≠ ⊥ then
13 |   |   |   |   |   return ⊥
14 |   |   |   |   end
15 |   |   |   |   reach := reach ∨ cex'.r
16 |   |   |   end
17 |   |   |   c := c ∨ reach
18 |   |   end
19 |   else
20 |   |   c := c ∧ ¬preds
21 |   end
22 end
23 return c
```

3.2.1 The Algorithm NSMC

Algorithm 1 shows the pseudo code of our main algorithm NSMC. The inputs of this algorithm are the two Mealy transducers M_1 and M_2 of the C++ classes, a correspondance mapping (δ, Δ) of M_1 and M_2, and an initial candidate invariant c. After execution, the algorithm will either return an invariant that enabled an inductive proof of equivalence or $\bot$ if the models are not equivalent.

In line 1, we slightly modify the initial candidate invariant to ensure that it is fulfilled for all initial states. With this change, we can be sure that if the candidate invariant is not fulfilled for an initial state, it is due to a modification that we did and a detected counterexample is reachable.

We exclude non-equivalent states and their predecessors from the candidate invariant in line 2. If any unsafe state is reachable, the two models are not equivalent. If one such state is reachable, calling a pair of methods from Δ in a reachable state would return different results. This step is realized by calling the function PRED-STATES with the initial candidate invariant as pre-hypothesis and "true" as the post-hypothesis. The function will return a formula that describes all originating states of counterexamples under the given hypotheses as well as their predecessors that fulfill the pre-hypothesis. The post-hypothesis "true" is valid for all states. This means that counterexamples due to reached

states that do not fulfill the post-hypothesis do not exist and every generated counterexample is generated due to non-equivalent output.

If an initial state $s_0 \in \delta$ is excluded from the candidate invariant in this step, an unsafe state is reachable and the models are proved to be not equivalent which is returned in line 3.

In the following loop that starts in line 4 the remaining counterexamples are handled. Each of these counterexamples arises from a reached state that does not fulfill the candidate invariant because counterexamples due to non-equivalent output were already removed from the candidate invariant in the previous step.

With such a counterexample $cex = (o, r, m)$, we generate the formula *preds* which describes all predecessors of r that fulfill the hypothesis in line 5. This is realized by calling PRED-STATES with the current candidate invariant as pre-hypothesis and the post-hypothesis $\neg r$. Thus all states that fulfill *preds* are predecessors of r.

If an initial state $s_0 \in \delta$ fulfills *preds*, r is reachable but does not fulfill the candidate invariant. This is checked in line 6. Since r does currently not fulfill the candidate invariant, it needs to be checked if it is an unsafe state. If r is an unsafe state, the models are not equivalent because an unsafe state is reachable which is returned in line 8. Otherwise, the candidate invariant is modified to fulfill r and possible descendants of r that do not fulfill c. This is realized by generating a formula *reach* in line 10 that initially describes the assignment of r. If a state r' exists, that can be reached from any state that fulfills *reach*, but does fulfill neither c nor *reach*, it is checked in line 12 if that state is safe. If it is unsafe, M_1 and M_2 are not equivalent which is returned in line 13. Otherwise, *reach* is modified in line 15 to fulfill r'. Once we cannot find any more reachable states that do not fulfill the candidate invariant, we modify the candidate invariant in line 17 to include all these reachable states. This modification does not allow any unsafe states in c as all states that fulfill *reach* have been checked and are guaranteed to be safe.

If the initial state does not fulfill *preds*, we can safely remove *preds* from the candidate invariant in line 20.

This loop from line 4 to 22 is repeated until no counterexamples remain. When the equivalence was not disproved until the end of the loop, the models are equivalent since no counterexamples remain to invalidate the equivalence which is returned in line 23. The final candidate invariant is an invariant of the lockstep machine that enabled an inductive proof and can be used to speed up further verification.

3.2.2 The Algorithm PRED-STATES

An essential function that is used by our algorithm is PRED-STATES. Its inputs are two Mealy transducers M_1 and M_2, a method mapping Δ, and a pre- and post-hypothesis h_{pre} and h_{post}. It returns a formula that describes all originating states of counterexamples for the given models and hypotheses as well as a subset of their predecessors. The subset contains all predecessors p where a path from p to an originating state of a counterexample exists that only contains states that fulfill the pre-hypothesis. Algorithm 2 shows the pseudo code of PRED-STATES.

The return value *pred* of PRED-STATES is initialized with "false" in line 1, which describes the empty set of states. The loop that starts in line 2 continues

Algorithm 2: PRED-STATES

input : two Mealy transducers M_1 and M_2 of C++ classes, a method mapping Δ, and two hypotheses h_{pre} and h_{post}
output : a logic formula *pred* that describes all starting states of counterexamples and their predecessors that fulfill h_{pre}

```
1 pred := false;
2 while (cex := Check(h_pre ∧ ¬pred, h_post ∧ ¬pred, M_1, M_2, Δ)) ≠ ⊥ do
3 |   gen := generalize(cex, M_1, M_2, Δ, h_pre, h_post);
4 |   pred := pred ∨ gen;
5 end
6 return pred
```

Algorithm 3: GENERALIZE

input : a clause c that is a negation of an assignment to all variables and two hypotheses h_{pre} and h_{post}
output : a formula that describes the generalized clause c

```
1 c' := REMOVE-DC(c, h_pre, h_post)
2 C := {c'} ∪ CHECK-EQUALS(c', h_pre, h_post)
3 C := C ∪ CHECK-INTERVALS(c', h_pre ∧ C, h_post)
4 return ⋀_{c̃∈C} c̃
```

while counterexamples still exist. If there are still counterexamples to the pre-hypothesis $h_{\text{pre}} \wedge \neg pred$ and the post-hypothesis $h_{\text{post}} \wedge \neg pred$, one is generated in line 2. The used pre-hypothesis enables us to prevent counterexamples with originating states that are already described by *pred*. The post-hypothesis is not fulfilled for states in *pred*. Thus, predecessors of states that fulfill *pred* will be detected as originating states of counterexamples and enable us to detect all predecessors of counterexamples. After detecting a counterexample, it is generalized in line 4 by calling `generalize`. This function allows us in each step to consider multiple similar states instead of only a single counterexample. The result of `generalize` is a formula that describes similar originating states of counterexamples to the given pre- and post-hypothesis. The generalized formula is added to *pred* in line 4. Now, *pred* is fulfilled for all detected originating states of counterexamples. The loop is repeated until no counterexamples remain. Finally, *pred* is returned in line 6.

3.2.3 The algorithm GENERALIZE

Algorithm 3 shows the pseudo code of the algorithm GENERALIZE. It is used to generalize a counterexample, so similar assignments can be considered at the same time. The algorithm receives a clause c as input. The clause c is a negated assignment of values to all variables of the two models. Furthermore, the pre- and post-hypothesis h_{pre} and h_{post} that were used to generate the assignment are given as well. The return value is a set of clauses C that describes the generalized clauses based on c.

In a first step in line 1, c is modified by removing all irrelevant assignments. The remaining assignments suffice to cause a counterexample under h_{pre} and

Algorithm 4: REMOVE-DC

input : a clause c that is a negated assignment to all variables and two hypotheses h_{pre} and h_{post}
output : a generalized clause c'

1 $V := \mathsf{Variables}(M_1) \cup \mathsf{Variables}(M_2)$
2 $J := \emptyset$
3 $a := \neg c$
4 **foreach** $v \in V$ **do**
5 $\quad S := V \setminus (J \cup \{v\})$
6 $\quad a' := \bigwedge_{\hat{v} \in S} \hat{v} \equiv a(\hat{v})$
7 $\quad$ **if** $\mathsf{CheckCex}(h_{pre} \wedge a', h_{post}, M_1, M_2, \Delta) = \bot$ **then**
8 $\quad\quad J := J \cup \{v\}$
9 $\quad$ **end**
10 **end**
11 **return** $\neg \bigwedge_{v \in V \setminus J} v \equiv a(v)$

h_{post}, no matter which values are assigned to the remaining variables.

Next, we check if variables that still have assignments and are different in the counterexample always need to be equal in line 2. If variables are different in a counterexample, the counterexample could be caused by the difference of the variables, meaning that these variables need to be equal. According clauses that describe the equality are added to C.

The last used heuristic checks if variables can be restricted to a certain interval. The clauses that describe a new lower or upper bound are added to C in line 3. Finally, C is returned in line 4.

This specific order of generalization was chosen as each step lessens the effort of the next one. When we remove all irrelevant assignments before checking for equal pairs of variables, we only consider the relevant variables. Checking for irrelevant variables would not generalize further as the value of the irrelevant variables do not matter. Furthermore, when we detect equal variables before checking for intervals, we can use this information for better initial upper or lower bounds as equal variables share the same interval.

During generalization, the function CheckCex is regularly called. This function receives the same inputs as Check, but checks if all states that fulfill the pre-hypothesis cause a counterexample for at least one possible method call, i.e., either reach a state that does not fulfill the post-hypothesis or generate different outputs.

Removing Don't-Care Assignments

In the algorithm REMOVE-DC, shown in Algorithm 4, we try to remove assignments to don't-care variables from c as these are not relevant for the counterexample and thus consider similar assignments as well. The input c is a negated counterexample and as such a negated assignment to all variables. In lines 1 – 3, we initialze the set V with all variables of the two models M_1 and M_2, the set J as empty set, and the formula a as a negation of c. The set J will be used to store all irrelevant variables later on. The formula a describes the assignment that is negated in c.

Algorithm 5: CHECK-EQUALS

input : a clause $c \in C$ that provides a negated partial assignment to variables, and two hypotheses h_{pre} and h_{post}
output : a set C of clauses

```
1  V_1 := Variables(M_1)
2  V_2 := Variables(M_2)
3  a := ¬c
4  C := ∅
5  foreach (v_1, v_2) ∈ V_1 × V_2 do
6      if a(v_1) = a(v_2) then continue
7      if a(v_1) = ⊥ or a(v_2) = ⊥ then continue
8      if δ(v_1) ≠ δ(v_2) then continue
9      e := (v_1 ≡ v_2)
10     if Check(h_pre ∧ e, e, M_1, M_2, Δ) = ⊥ then
11         C := C ∪ {e}
12         continue
13     end
14     if CheckCex(h_pre ∧ e, h_post, M_1, M_2, Δ) = ⊥ then
15         C := C ∪ {e}
16     end
17 end
18 return C
```

In the loop from lines 4 to 10, we check for each variable $v \in V$ if v is relevant for the counterexample. Initially, we prepare the set $S = V \setminus (J \cup \{v\})$. As such, S does not contain the variables that have already shown to be irrelevant and does not contain v as well. This enables us to check if the assignments of all other variables suffice to cause a counterexample. We then prepare an according assignment a' in line 6. If a' together with h_{pre} suffices to always cause a counterexample, v is added to J in line 8.

As this is a greedy approach, it does not guarantee a minimal set of relevant variables. However, it only takes linear time to compute a usually sufficient solution. This process could be speed up by analyzing the proof that generated the original counterexample like it is done in IC3 [13], an implementation of PDR. They analyze the proof that generated the counterexample and can directly categorize variables that were not used in the proof as don't care. In our case, this would entwine our algorithm further with the underlying model checker and would decrease the ability of our algorithm to easily exchange the model checker.

Finally, we return a new negated assignment in line 11, that contains all variables, that are not in J.

Detecting Equal Variables

Algorithm 5 shows the algorithm CHECK-EQUALS. It checks for variables within the two models that may be required to be equal. As the input, we get a clause c which is the result from Algorithm 4 and as such a negated partial assignment, and finally the hypotheses h_{pre} and h_{post}. In this algorithm, we want to find clauses that state equality between variables to block more counterexamples.

In lines 1 and 2 we get the sets V_1 and V_2 that contain all variables from the first and the second model, respectively. In line 3, we negate c to get the non-negated assignment a. We initialize the set of clauses C as empty set in line 4.

Next, we check for each pair of variables $(v_1, v_2) \in V_1 \times V_2$ if the counterexample could be caused by the inequality of v_1 and v_2. First, we check for some requirements in lines 6 – 9 before we consider equality. If the variables are equal in the counterexample, the counterexample cannot be caused by their inequality. If v_1 or v_2 is not assigned in a, it was detected as irrelevant for the counterexample by the previously executed algorithm REMOVE-DC and as such the inequality is also not relevant. This allows us to skip all checks for equality that contain irrelevant variables and focus our algorithm only on pairs of relevant variables. Finally, when the variables are not equal in the initial state, we cannot add equality to C as the variables are obviously not equal in all reachable states.

Next, we initialize the formula e which states that v_1 and v_2 are equal in line 10. The check in line 11 verifies, if all states that fulfill the $h_{\text{pre}} \wedge e$ can only reach states where $v_1 \equiv v_2$. In that case, we can safely add e to C and check for the next pair of variables. Otherwise, we check if all states that fulfill the pre-hypothesis but have different values for v_1 and v_2 cause counterexamples. In this case, we can also add e to C as we only block states that are unsafe or can break the post-hypothesis.

Finally, in line 19, we return the modified set C.

Limiting Variables to Intervals

The algorithm CHECK-INTERVALS is used to limit variables to certain intervals instead of blocking single values and is shown in Algorithm 6. As in Algorithm 5, we get a negated partial assignment c and two hypotheses h_{pre} and h_{post} as inputs. As output, we return a set of clauses that limits variables to certain intervals to block additional counterexamples.

Initially, we get the assignment a that is negated in c in line 1. For each integer variable v that is assigned in a, we determine the upper bound u and lower bound l according to the pre-hypothesis in lines 5 and 6. The upper and lower bounds are detected by looking for terms within the pre-hypothesis that limit v or variables that are equal to v. While this does not guarantee the optimal bounds that could be deduced from the hypothesis, it is done quickly and suffices for our approach as we merely use these bounds as starting points. Moreover, when new bounds are learned from this algorithm, they are detected in further iterations.

We try to decrease u by replacing it with $a(v)$. The value $a(v)$ must be less than u because the assignment of a needs to fulfill the pre-hypothesis. The decrease is valid if all assignments to v outside of the interval between l and *val* would lead to counterexamples, as checked in lines 8 – 10, or the value of v in the initial state is within the interval and the value of v remains within the interval from any state fulfilling the pre-hypothesis after calling any function, as checked in lines 11 – 13. In a next step, we try to increase the lower bound analogously. This preparatory step is done to consider cases where $a(v)$ lies outside of an optimal interval for v, which is one possibility for a counterexample. In these cases, the interval can be shrinked by a significant amount with only few checks. Otherwise, these checks will only increase the runtime slightly.

Algorithm 6: CHECK-INTERVALS

input : a clause c which is a negated partial assignment and two hypotheses h_{pre} and h_{post}
output : a set C of clauses

1 $a := \neg c$
2 $V := \{v | a(v) \neq \bot\}$
3 $C := \emptyset$
4 **foreach** $v \in V$ **do**
5 $l := \mathsf{lowerBound}(v, h_{\text{pre}})$
6 $u := \mathsf{upperBound}(v, h_{\text{pre}})$
7 $decreaseUp =$ false
8 **if** $\mathsf{CheckCex}(h_{pre} \wedge (v \geq a(v)), h_{post}, M_1, M_2, \Delta) = \bot$ **then**
9 $decreaseUp =$ true
10 **end**
11 **if** $\mathsf{Check}(h_{pre} \wedge (v < a(v), h_{post} \wedge (v < a(v)), M_1, M_2, \Delta) = \bot$ **then**
12 **if** $\delta \rightarrow (v < a(v))$ **then** $decreaseUp =$ true
13 **end**
14 **if** $decreaseUp$ **then** $u := a(v)$
15 //Analogous process for lower bound
16 ...
17 **if** $increaseLow$ **then** $l := a(v)$
18 $stop =$ false
19 **while** $\neg stop$ **do**
20 $s := 1$
21 $decreaseUp =$ false
22 **if** $\mathsf{CheckCex}(h_{pre} \wedge (v \geq (u - s)), h_{post}, M_1, M_2, \Delta) = \bot$ **then**
23 $decreaseUp =$ true
24 **end**
25 $pre := h_{\text{pre}} \wedge (l < v < (u - s))$
26 $post := h_{\text{post}} \wedge (l < v < (u - s))$
27 **if** $(\delta \rightarrow (v < (u - s))) \wedge \mathsf{Check}(pre, post, M_1, M_2, \Delta) = \bot$ **then**
28 $decreaseUp =$ true
29 **end**
30 **if** $decreaseUp$ **then**
31 $u := u - s$
32 $s := 2s$
33 **else**
34 **if** $s \neq 1$ **then**
35 $s := \frac{s}{2}$
36 **else**
37 $stop =$ true
38 $C := C \cup \{v < u\}$
39 $h_{\text{pre}} = h_{\text{pre}} \wedge (v < u)$
40 **end**
41 **end**
42 **end**
43 $stop =$ false
44 //Analogous process for lower bound
45 ...
46 **end**
47 **return** C

After using this coarse approach for some initial shrinking of the interval, we try to shrink the intervals even further by decreasing the detected upper bound of the interval and increasing the lower bound. For decreasing the upper bound, we initialize a step size s with 1 in line 20. Then, we check if it is possible to decrease u by s in lines 21 – 29. We use the same checks as before, i.e., checking if all values outside the new interval are counterexamples or it is impossible to leave the interval. When this is possible, we decrease u by s and double s in lines 31 and 32 and check again with the new values. We double s to decrease the interval within logarithmic instead of linear time. When the decrease is not possible and s is not 1, we halve s in line 35 and check again. Otherwise, s is 1, meaning u is a valid upper bound, but $u - 1$ is not. Thus, we have detected an optimal upper bound for v, stop the loop in line 37, and add $(v < u)$ to C and the pre-hypothesis in lines 38 and 39. After decreasing u, l is increased analogously. The idea behind starting with small step sizes is to quickly refine existing intervals that are slightly off. For example, if an existing interval is off by one, this is quickly detected by the first decrease.

3.2.4 Sketching correctness

In this section, we will sketch why NSMC does always terminate and decides equivalence correctly. The only requirement for this property is that the underlying model checker also always terminates and decides correctly which we will assume for the remainder of this section. Based upon this sketch, a full proof is possible. But as it needs to consider multiple corner cases it is too long for this section.

For the proof, the two Mealy transducers $M_1 = (S_1, S_{0_1}, X_1, Y_1, \phi_1, \psi_1)$ and $M_2 = (S_2, S_{0_2}, X_2, Y_2, \phi_2, \psi_2)$ describe the analyzed systems. The resulting lockstep machine of M_1 and M_2 is $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$ and (δ, Δ) is any correspondence mapping of M_*. When paths are discussed during this proof, we only consider paths under inputs from Δ. Let $U \subseteq S_*$ describe all unsafe states in M_* and $C \subseteq S_*$ describe all states that fulfill the current candidate invariant c.

First, we will argument why NSMC always decides equivalence correctly, i.e., decides that M_1 and M_2 are equivalent under (δ, Δ) iff there exists no path from any $s_0 \in \delta$ to a state $u \in U$. Algorithm 1 has four possibilities to terminate in line 3, line 8, line 13, and line 23. In line 3, the algorithm decides that the models are not equivalent if $\delta \rightarrow \neg c$, i.e., $\delta \setminus C \neq \emptyset$. As the first line adds all initial states to the candidate invariant, $\delta \setminus C = \emptyset$ holds after that line. Thus, $\delta \setminus C \neq \emptyset$ is only possible in line 3 if an initial state $s_0 \in \delta$ was excluded from the candidate invariant in line 2. This would mean, that s_0 is a predecessor from a state $u \in U$ or $s_0 \in U$. In both cases the models are not equivalent as a path from s_0 to an unsafe state exists which the algorithm decides correctly.

The return-statements in line 8 and 13 are both contained within an if-block from line 6 which states that $\delta \rightarrow \mathit{preds}$. The formula *preds* was generated before and contains predecessors that fulfill the candidate invariant of the reached state *cex.r* that does not fulfill the candidate invariant. Thus, $\mathit{cex.r} \notin C$. Let *preds* define the set $P = \{s \in S_* | \mathit{preds}(s)\}$. Since $\delta \rightarrow \mathit{preds}$, $\delta \cap P \neq \emptyset$ and therefore the state *cex.r* is reachable. In line 7, we check if *cex.r* is a safe state and return $\perp$ if it is not. As $\mathit{cex.r} \in U$ and *cex.r* is reachable, a path from an initial state $s_0 \in \delta$ to $\mathit{cex.r} \in U$ exists and the decision that the models are not equivalent

is correct. Similarly, in line 13, we check if a state that is reachable from *cex.r* is unsafe and return $\perp$ in that case. Since a path from *cex.r* to an unsafe state $u \in U$ exists, there also exists a path $s_0, \ldots, \mathit{cex.r}, \ldots, u$ from an initial state $s_0 \in \delta$ to u. Thus, deciding that the models are not equivalence is correct.

The final possibility for NSMC to terminate is in line 23 when the current candidate invariant is returned, meaning the models are equivalent and c is an inductive invariant. This is correct as line 23 can only be reached if the while-loop from line 4 to 22 terminates which means that there exists no counterexample. Thus, every transition from a state in $s_1 \in C$ leads to a state $s_2 \in C$. Furthermore, C contains no unsafe states, i.e., $C \cap U = \emptyset$. In addition, $\delta \subseteq C$, as initial states cannot be removed after line 2 and the algorithm would have returned $\perp$ in line 3 if $\delta \subseteq C$. Thus, states within C can only reach states in C. Since $\delta \subseteq C$, all initial states in δ can only reach states in C. Since $C \cap U = \emptyset$, initial states from δ cannot reach unsafe states in U, meaning the models are equivalent and the candidate invariant c that defines C is an inductive invariant as returned by the algorithm.

While we have argued that NSMC does always decide correctly when it terminates, we still need to show that the algorithm always terminates. The main obstacle that could prevent termination of the algorithm is the while-loop from line 4 to 22. In this loop, the candidate invariant c and therefore the set C is modified to consider counterexamples of states that do not fulfill the candidate invariant but are reachable from states that fulfill the candidate invariant.

If the algorithm does not terminate during an iteration of the loop execution, the candidate invariant is changed afterwards to either include more states in line 17 or exclude some states in line 20. In addition, if a state $\tilde{s}$ was added to C in line 17, it was reachable from an initial state $s_0 \in \delta$ through a path $s_0, s_1, \ldots, s_n, \tilde{s}$, where $s_0, s_1, \ldots, s_n \in C$. If $\tilde{s}$ was to be removed in a later iteration of the loop, it needs to be in the set $P = \{s \in S_* | \mathit{preds}(s)\}$ in line 5. However, as the path $s_0, s_1, \ldots, s_n, \tilde{s}$ exists, $s_0, s_1, \ldots, s_n \in P$ as s_n is a predecessor of $\tilde{s}$ and each s_i is a predecessor of s_{i+1} for $0 \leq i < n$. As such, $\delta \cap P \neq \emptyset$ and $\delta \rightarrow \mathit{preds}$ is fulfilled. Thus, $\tilde{s}$ cannot be removed from C after is was added in a previous iteration of the loop. As such, each state $s \in S_*$ can at most be added once to C and therefore also be removed at most once. Thus, the number of modifications that can be done to the candidate invariant within the loop is at most $2|S_*|$. As S_* is a finite set, the algorithm needs to terminate.

As we have discussed that NSMC always terminates and always decides correctly, it is always able to decide equivalence.

3.3 Using PDR in our algorithm EASY

As PDR [7, 13] has been shown to be an effective model checking technique, it could support the equivalence check done in NSMC. As equivalence checking is a specific case of model checking, the techniques used by PDR can be utilized for an equivalence check as well. However, as we use logical formulas over the member variables of C++ classes instead of boolean variables only, our model is different but can be used in a PDR-like algorithm. This idea is implemented in the algorithm EASY, which will be able to decide equivalence within a much shorter time in some cases where the PDR-like implementation supports the equivalence check and generates only a small overhead in those cases where it

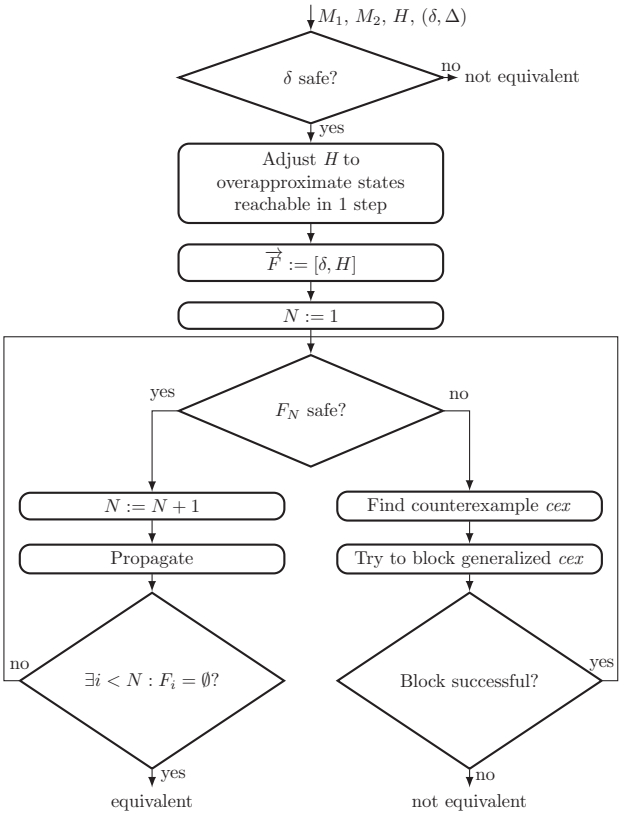


Figure 3.4: Checking equivalence between M_1 and M_2 under (δ, Δ)

does not. Especially underapproximations profit from PDR as we can easily drop clauses that are not fulfilled for all reachable states while still keeping the other detected or given clauses. On the other hand, we keep the strengths of NSMC like the ability to provide a candidate invariant or generalizing counterexamples. Thus, we use the basic construct of PDR but include additional features. Like NSMC, a candidate invariant can be used to speed up the equivalence check. The generalization of counterexamples uses additional heuristics to compute equal variables and intervals for integer variables. As NSMC and EASY consider different types of variables and consider relations that are specific to those types, e.g., upper and lower bounds of integer variables, we can generate additional helpful clauses compared to PDR that only computes don't-care bits. Therefore, our algorithms can handle all kinds of clauses while PDR solely focuses on cubes, i.e., partial assignment of the boolean bits.

Beside the use of a PDR-like approach, EASY also uses a new algorithm to determine the intervals of variables during generalization, which allows shorter runtimes.

The algorithm EASY decides for two models, their correspondence mapping, and a candidate invariant. The candidate invariant can be "true" which would provide the algorithm no additional knowledge about the models. It can be given manually by the developer, who should have detailed knowledge, or can be generated by third-party-tools.

The basic algorithm is sketched in Figure 3.4. We start by checking if the initial states δ of the lockstep machine are safe. If they are not safe, the models are not equivalent.

If they are safe, we check if the candidate invariant H overapproximates all states that are reachable in one step by executing methods from Δ. If this is not the case, we adjust H and remove clauses that are not fulfilled by all states that are reachable in one step. This adjustment is important to ensure, that the initial clause vector $\overrightarrow{F} = [\delta, H]$ fulfills property 3.2. Finally, we initialize N with 1, as this is currently the index of the last element of $\overrightarrow{F}$. Next, we check if all states in F_N are safe.

Unless all states in F_N are safe, there exists a counterexample *cex* that describes an unsafe state. We try to block *cex* and similar assignments by adding according clauses to $\overrightarrow{F}$ without breaking property 3.2. If this is not successful, the detected unsafe state is reachable and the models are not equivalent. Otherwise we check again, if all states in F_N are safe.

If they are, we increase the length of $\overrightarrow{F}$ by 1 and try to propagate the clauses as far towards F_N as possible. This is done by checking for each clause c in a set F_i if that clause is still fulfilled when calling any method in a state that fulfills $F_i\uparrow$. In that case, we can safely move c from F_i to F_{i+1} without breaking property 3.1. If there exists an F_i within the clause vector that is empty afterwards, we have proven equivalence and the algorithm terminates. Since F_i is empty, $F_i\uparrow = F_{i+1}\uparrow$. As all states within $F_i\uparrow$ can only reach states that fulfill $F_{i+1}\uparrow = F_i\uparrow$ by property 3.2, $F_i\uparrow$ is an invariant. By property 3.1, all states within $F_i\uparrow$ are safe and thus all reachable states are safe and the models are equivalent. If there is no empty set, we check again if the newly generated F_N describes safe states by checking if all states within F_N are safe. This is repeated until the algorithm terminates by deciding equivalence.

Example 2. Consider the counters from Example 1. Both counters are initialized with 0, i.e., $\delta = \{(0,0)\}$, and both counters use the function `countUp`, i.e., $\Delta = \{\texttt{countUp}, \texttt{countUp}\}$. We want to provide a candidate invariant to speed up the process. We try to provide an upper limit for the counter, but chose faulty values. In addition, we do not add equality to the candidate invariant. This results in the bad candidate invariant $i = mod \leq 2 \wedge if \leq 2$, where *mod* and *if* denote the member variables `count` of two counter implementations, respectively. This results in two clauses for the set H: $mod \leq 2$ and $if \leq 2$.

When we start the algorithm, we check if all initial states in δ are safe. As both counters return 1 when `countUp` is called in the initial state, they are safe.

Next, we check if all states that are reachable within one step fulfill the candidate invariant. As the counter can only count up to 1 within one step, the candidate invariant holds in these states and we do not modify H. Then, $\overrightarrow{F}$ and N are initialized and we check if all states in F_N are safe. This is not the case and we find a counterexample with an originating state $cex = (mod \equiv 1 \wedge if \equiv 2)$. As *cex* is not reachable from the initial state, we can add $\neg cex$ to F_1 or generalize *cex* and add $(if \equiv mod)$ such that

$$F_1 = \{mod \leq 2, if \leq 2, \neg(mod \equiv 1 \wedge if \equiv 2), if \equiv mod\}.$$

When checking again, we realize that F_N is safe now and increase N by 1. We then try to propagate clauses. Executing `countUp` in a state that fulfills $F_1\uparrow$ leads to a state where both counters are equal and the detected counterexample is not fulfilled. Thus, we can move $(mod \equiv if)$ and $\neg(mod \equiv 1 \wedge if \equiv 2)$ to F_2. However, we cannot move $mod \leq 2$ and $if \leq 2$ as we can reach counter values above 2.

When we check if the states within $F_N = F_2$ are safe, we find the originating state $cex = (mod \equiv 6 \wedge if \equiv 6)$ of a counterexample. Since we already know that the values are equal, we can safely remove one assignment from the counterexample, resulting in $cex = (mod \equiv 6)$. We cannot reach a state that fulfills *cex* from a state that fulfills $F_1\uparrow$. Thus, we can add $\neg cex$ to F_2. In addition, we generalize *cex* and can even add $mod \leq 3$ to F_2, resulting in

$$F_2 = \{\neg(mod \equiv 1 \wedge if \equiv 2), mod \equiv if, \neg(mod \equiv 6), mod \leq 3\}.$$

Now, F_2 describes exactly all reachable states of the lockstep machine. We find out that all states within F_N are safe and try to propagate clauses. Since for each clause c in F_2, that clause is fulfilled after calling any function from a state that fulfills $F_2\uparrow$, we can move c from F_2 to F_3. Since we can move all clauses, $F_2 = \emptyset$ afterwards and we return that the models are equivalent under the invariant $F_2\uparrow$.

In the following Sections 3.3.1 to 3.3.4, we will describe the algorithm, starting at the top level and decreasing the level of abstraction with each section. In the final Section 3.3.5, we will discuss features of the algorithm.

3.3.1 Top Level Algorithm

The top level algorithm of EASY is shown in Algorithm 7. It decides if two C++ classes given as Mealy transducers M_1 and M_2 are equivalent under a correspondence mapping (δ, Δ). To speed up the algorithm, a candidate invariant

Algorithm 7: EASY-PDR

input : two Mealy transducers M_1 and M_2 of C++ classes, a correspondence mapping (δ, Δ), and a set of expressions H that describes the candidate invariant

output : an invariant if M_1 and M_2 are equivalent under (δ, Δ) or $\perp$ otherwise

```
1  //Check if initial states are safe
2  if Check(δ, true, M1, M2, Δ) ≠ ⊥ then
3  |  return ⊥
4  end
5  //Check candidate invariant for first step
6  cex := Check(δ, H, M1, M2, Δ)
7  while cex ≠ ⊥ do
8  |  //Weaken candidate invariant if needed
9  |  H := H \ {c ∈ H | c blocks cex}
10 |  cex := Check(δ, H, M1, M2, Δ)
11 end
12 //F0 are the initial states
13 F⃗.push(δ)
14 //F1 is the candidate invariant
15 F⃗.push(H)
16 N := 1
17 while true do
18 |  //Check for unsafe states
19 |  cex := Check(FN, true, M1, M2, Δ)
20 |  if cex ≠ ⊥ then
21 |  |  //Recursively block the counterexample
22 |  |  if ¬ BLOCK(TClause(¬cex.o, N), F⃗) then
23 |  |  |  return ⊥
24 |  |  end
25 |  else
26 |  |  //A new frame and propagating clauses
27 |  |  F⃗.push(∅)
28 |  |  N := N + 1
29 |  |  F⃗ := PROPAGATE(F⃗, N)
30 |  |  if ∃i < N : Fi = ∅ then
31 |  |  |  return Fi↑
32 |  |  end
33 |  end
34 end
```

is given as input as well. If the models are equivalent, an invariant is returned. Otherwise, the algorithm returns $\perp$. In this case, F_N contains a reachable counterexample and $\overrightarrow{F}$ describes a way to reach that counterexample.

In the beginning of the algorithm, we check if the initial states δ given by the correspondence mapping are safe. This is done in line 2 by using Check from Section 3.2.3. If a non-safe initial state exists in δ, the models cannot be equivalent and $\perp$ is returned in line 3.

In lines 5 – 11, it is checked if the candidate invariant overapproximates all states that are reachable from an initial state by executing a single pair of functions from Δ. If a counterexample *cex* is found, we weaken the candidate invariant by removing all clauses that are not fulfilled for the detected assignment *cex*.

After these initial checks, we initialize the vector $\overrightarrow{F}$ by pushing the initial states from δ as F_0 in line 13 and afterwards the eventually weakened candidate invariant H as F_1 in line 15. In line 16 we initialize N with 1. N describes the last index of $\overrightarrow{F}$.

The following loop in lines 17 – 33 will refine the candidate invariant until equivalence is proven or a real counterexample to equivalence is found.

First, we check if the current approximation $F_N\uparrow$ contains only safe states in lines 19 and 20. If an unsafe counterexample *cex* exists, we try to recursively block the detected assignment by calling the algorithm BLOCK which is described in Section 3.3.2. The input of BLOCK is timed claues *TClause* that contains information about the clause that needs to be blocked as well as the frame in which it needs to be blocked. In this case, the clause is the negated counterexample and the frame is N since we detected the unsafe state in F_N If BLOCK does not succeed in blocking, *cex* describes a reachable non-safe state and M_1 and M_2 are not equivalent. This is returned in line 23. Otherwise, *cex* and similar assignments are blocked in F_N and will not trigger again.

If no unsafe states exist in F_N, we have proven, that no safe state is reachable in N steps. We add another frame to consider states that are reachable in $N+1$ steps. For this additional frame, we push an empty set to $\overrightarrow{F}$ and increase N by 1 in lines 27 and 28. Next, the algorithm PROPAGATE described in Section 3.3.3 is used to move clauses within $\overrightarrow{F}$ as close to F_N as possible.

If an empty set F_i with $i < N$ exists after the propagation, the models are equivalent because $F_i\uparrow = F_{i+1}\uparrow$. By definition, all states that fulfill $F_i\uparrow$ can only reach states that fulfill $F_{i+1}\uparrow = F_i\uparrow$, so $F_i\uparrow$ is an overapproximation of all reachable states. In addition, F_i only contains safe states as $i < N$. Thus, the models M_1 and M_2 are equivalent and $F_i\uparrow$ is returned as invariant in line 31.

3.3.2 Blocking Unsafe States Recursively

The algorithm BLOCK is used to block a detected counterexample in $\overrightarrow{F}$. The counterexample is given as a TClause *c0*, that contains a clause *c0.clause* and a frame number *c0.frame*. The clause *c0.clause* describes the negated assignment of the counterexample and the number *c0.frame* describes the element of $\overrightarrow{F}$ where *c0.clause* needs to be blocked. The algorithm returns "true" iff the counterexample was successfully blocked and all states that are reachable within *c0.frame* steps fulfill *c0.clause*.

Algorithm 8: BLOCK

input : a TClause *c0* that contains a clause *c0.clause* and a number *c0.frame* where *c0.clause* needs to be fulfilled in *c0.frame* and a clause vector $\overrightarrow{F}$

output : a Boolean value that is true iff the counterexample was blocked

```
1  PrioQ<TClause>Q
2  Q.add(c0)
3  while Q.size() > 0 do
4      c := Q.popMin()
5      f := c.frame, cl := c.clause
6      //Detected a real counterexample?
7      if f = 0 then return false
8      if ¬ follows(c, F_f↑) then
9          C := GENERALIZE(cl, F_f↑, F_{f+1}↑)
10         //Is C following from the previous frame?
11         cex := Check(F_{f-1}↑, C, M_1, M_2, Δ)
12         if cex = ⊥ then
13             F_f := F_f ∪ C
14         else
15             //cex and C need to be checked
16             Q.add(TClause(¬cex.o, f − 1))
17             foreach c' ∈ C do
18                 Q.add(TClause(c', f))
19             end
20         end
21     end
22 end
23 return true
```

The algorithm uses a priority queue Q that is initialized in line 1. In line 2, we add *c0* to the queue. While Q is not empty, we pop one element c of Q with the lowest frame number *c.frame* in line 4 and initialize the variables f and cl as *c.frame* and *c.clause*, respectively in line 5. If *c.frame* is 0, we have detected a reachable counterexample, as the generated clauses describe a path that leads from an initial state to a state that fulfills *c0.clause* and we return "false" in line 7.

Otherwise, we check if c follows from the clauses of its current frame $F_f\uparrow$. This is the case if *c.clause* follows from $F_f\uparrow$. If c follows, we do not need to analyze it further as we know that states that are reachable in f steps fulfill *c.clause*. If c does not follow, we generalize c to also consider similar assignments in line 9 and get a set C of clausesby using the algorithm GENERALIZE described in Section 3.2.3.

Next, we check if C follows from the previous frame $f-1$ in line 11. If all executions of methods in states that fulfill $F_{f-1}\uparrow$ lead to states that fulfill C, all states that are reachable within f steps must fulfill C. Otherwise, we need to check if the detected counterexample is blocked in the previous frame $f-1$ and add the according TClause to Q in line 16. As we did not show that states that are reachable in f steps fulfill C, we need to put the according TClauses back on Q in lines 17 – 19 to check them again after ensuring that *cex* is blocked.

When Q is empty, the loop terminates. The vector $\overrightarrow{F}$ has been modified to ensure that all states that are reachable within *c0.frame* steps fulfill *c0.clause* and the algorithm returns "true".

3.3.3 Propagating Clauses

Algorithm 9: PROPAGATE

input : a clause vector $\overrightarrow{F}$ and a number N that describes the size of $\overrightarrow{F}$
output : a clause vector $\overrightarrow{F}$ with propagated clauses

```
1 foreach i := 1, ..., N − 1 do
2 |   foreach c ∈ F_i do
3 |   |   if Check(F_i↑, c, M_1, M_2, Δ) = ⊥ then
4 |   |   |   F_{i+1} := F_{i+1} ∪ {c}
5 |   |   |   F_i := F_i \ {c}
6 |   |   end
7 |   end
8 end
9 return F
```

The algorithm PROPAGATE modifies $\overrightarrow{F}$ by moving clauses within the sets as far towards F_N as possible while keeping property 3.2. The algorithm is shown in Algorithm 9.

The outer loop is executed for each F_i except for F_0 because this is the special case of initial states and the last set F_N as a clause cannot be moved further than F_N. Starting with $i = 1$, we check for each clause $c \in F_i$, if we can move c to F_{i+1} in line 3. The clause c can be moved, if all states that are reachable in a single step under the pre-hypothesis $F_i\uparrow$ fulfill c. This is done in lines 4 and 5. Since c has been moved to F_{i+1}, it is possible to move c even further as F_{i+1} is checked in the next iteration of the loop.

Finally, the modified vector $\overrightarrow{F}$ is returned in line 9.

3.3.4 New Determination of Intervals

Beside the change of structure into a PDR-like form, EASY also uses a new implementation to determine the intervals of variables, used during generalization. The algorithm is shown in Algorithm 10. The experiments show that this implementation reduces the runtime on our examples. The interface to this function is identical to the version of NSMC. Thus, the inputs are a negated assignment of a counterexample and a pre- and post-hypothesis. The algorithm returns a set of clauses that describes upper and lower bounds for variables.

The algorithm starts by initializing a as the non-negated assignment of c in line 1 and then determines the set V of all variables that have assigned values in a in line 2. Next, the set C is initialized as empty set in line 3.

We check for each variable $v \in V$ if we can limit the variable to a certain interval. We start by getting the current upper and lower bound of v from h_{pre}, if there is any. In the following line 7 we initialize a formula *up* that states that the pre-hypothesis is fulfilled and v is exactly at its current upper bound. If all states that fulfill *up* lead to counterexamples, we can decrease the upper bound by at least 1 and start searching for an optimal upper bound.

In contrast to the method of NSMC, we use binary search to find an optimal upper bound. We prepare the search, by initializing the lower detected and the upper detected value for the new upper bound *ld* and *ud* in lines 9 and 10 with the current upper and lower bound. Next, we compute their average u' as the potential new upper bound and finally initialize a Boolean variable *done* with "false".

While *done* is not set, we keep searching for an optimal upper bound for v. We prepare two formulas *up1* and *up2* in lines 14 and 15. The formula *up1* states that the pre-hypothesis is fulfilled and v is not smaller than u'. If all states in *up1* would cause counterexamples, u' would be a possible upper bound. The second formula *up2* is similar, but states that v is not smaller than $u' - 1$. In the lines 16 and 17, we check if all states within *up1* and *up2* cause counterexamples, respectively. If all states in *up1* cause counterexamples, but those in *up2* do not, we found an optimal upper bound, which is checked in line 18. We add the formula $(v < u')$ to C, set the upper limit u to u', and set *done* to "true" as we are done with the search for an upper bound.

If *up1* and *up2* both only contain states that cause counterexamples, which is checked in line 23, u' is a possible upper bound, but can be further decreased. Since we know that u' is a valid upper bound, we set *ud* to u' in line 24 and compute u' as the new average of *ud* and *ld* in line 25.

Algorithm 10: CHECK-INTERVALS

input : a clause c which is a negated partial assignment and two hypotheses h_{pre} and h_{post}
output : a set C of clauses

```
1  a := ¬c
2  V := {v | a(v) ≠ ⊥}
3  C := ∅
4  foreach v ∈ V do
5      l := lowerBound(v, h_pre)
6      u := upperBound(v, h_pre)
7      up := h_pre ∧ (v ≡ u)
8      if CheckCex(up, h_post, M_1, M_2, Δ) = ⊥ then
9          ld := l
10         ud := u
11         u' := (ld + ud)/2
12         done := false
13         while ¬done do
14             up1 := h_pre ∧ ¬(v ≤ u')
15             up2 := h_pre ∧ ¬(v ≤ u' − 1)
16             c1 := CheckCex(up1, h_post, M_1, M_2, Δ)
17             c2 := CheckCex(up2, h_post, M_1, M_2, Δ)
18             if (c1 = ⊥) ∧ (c2 ≠ ⊥) then
19                 C := C ∪ {(v ≤ u')}
20                 u := u'
21                 done := true
22             end
23             if (c1 = ⊥) ∧ (c2 = ⊥) then
24                 ud := u'
25                 u' := (ud + ld)/2
26             end
27             if (c1 ≠ ⊥) then
28                 ld := ud
29                 u' := (ud + ld)/2
30             end
31         end
32     end
33     low := h_pre ∧ (v ≡ l)
34     if CheckCex(low, h_post, M_1, M_2, Δ) = ⊥ then
35         //Same checks for lower bound
36         ...
37     end
38 end
39 return C
```

Otherwise, if *up1* does not only contain counterexamples, an optimal upper bound is higher than u'. Similarly to the previous case, we increase *ld* by setting it to u' and compute the new average of *ud* and *ld*.

After we determined an upper bound for v, we do the analogous checks to determine a lower bound for v and finally return C in line 39.

3.3.5 Discussion

The described algorithm can easily learn new clauses by using the provided heuristics in the algorithm GENERALIZE. In the current implementation, equality of variables or certain intervals can easily be detected and speed up the decision if these kind of clauses can describe an optimal invariant, i.e., an invariant that suffices to show equivalence inductively.

Furthermore, if there are faulty clauses in the initial candidate invariant, these are left within the sets with lower index during propagation and are easily dropped from the final invariant.

Compared to NSMC, we do not consider a single logical formula, but handle a set of clauses. This allows a finer control over the current candidate invariant and enables actions like dropping problematic clauses, which is not possible in NSMC, where the algorithm would need to learn all problematic states instead, which causes a significant overhead up to non-feasible runtimes. To handle the clauses, the algorithm is structured like PDR.

Like PDR, we create an empty set in F_i when we have successfully detected an inductive invariant as PROPAGATE and BLOCK are similar to PDR with some adjustments to C++ setting and the initial candidate invariant. However, the algorithm GENERALIZE is different from PDR as we use different heuristics to generate insight into the modules while PDR generates cubes that describe partial assignment to the boolean variables. On the other hand, EASY considers different types of variables and considers relations that are specific to those types, e.g., upper and lower bounds of integer variables.

As further optimization, PROPAGATE, BLOCK, and GENERALIZE could easily be parallelized similar to PDR.

3.4 Experiments

In this section, we present our experiments with three examples and compare the performance of NSMC and EASY. The first example attempts to check functional equivalence of two counter designs similar to Example 1. The second example is dedicated to equivalence checking of an arithmetic unit. The third and final example is the most complex one and considers two models of processors, including arithmetic operations, reading from and writing to memory, as well as jump operations.

The experiments were conducted on a Lenovo T430 with an Intel Core i5-3320M CPU with 2.6GHz and 8GB of RAM running Windows 7 Professional 32bit. As model checker, we use CBMC v4.9 [8] and the logic formulas are represented as SMT2 formulas in Z3 v4.4.1 [10]. We consider a time limit of 6 hours for finding an inductive invariant and report **T/O** if no such invariant is found within this time limit.

3.4.1 Counter

Unlike the counters from Example 1, we use integer variables and increase the maximum value of the counter to 9,999,999 instead of 3. In addition, we also consider up to 10 parallel counters in each model that can be accessed individually. The increased maximum value directly increases the number of reachable states as well as the diameter of the corresponding Mealy transducer. The parallel counters will be used to show the scalability of our algorithms.

For the experiments on the counters, we consider different initial candidate invariants. The first candidate invariant *optimal* describes exactly all reachable states with

$$optimal = \bigwedge_{i \in \{1,\dots,\#\text{counters}\}} (mod_i \equiv if_i) \wedge 0 \leq mod_i \leq 9,999,999$$

where mod_i and if_i describe the i-th counter of the modulo- and if-counter, respectively. The first part of *optimal* describes that corresponding counter variables should have equal values. This is intuitively true, as these values are given as output and need to be equal for the output to be equal. The second part of the candidate describes that the counter values always range between 0 and 9,999,999, which is true as a counter is reset to 0 when it would reach 10,000,000. And since the counter always counts upwards, it is impossible to reach negative values.

The second candidate invariant is "true" which is the weakest possible overapproximation. This candidate is fulfilled for all possible states. Since the counters behave differently outside of the reachable range and cause faulty output if the counters store different values, the algorithms need to adjust this invariant to be true only for reachable states.

The final candidate invariant *boundTooLow* is similar to optimal but contains the faulty maximum value of 300 and thus underapproximates the reachable states:

$$boundTooLow = \bigwedge_{i \in \{1,\dots,\#\text{counters}\}} (mod_i \equiv if_i) \wedge 0 \leq mod_i \leq 300$$

While this candidate invariant is false for all non-reachable states, it is also false for most reachable states. As such, the algorithms need to loosen the candidate invariant to include all reachable states.

The experimental results in Figure 3.5 show, that equivalence can be shown within seconds when the candidate invariant *optimal* is used. However, EASY is slightly slower in these cases as some overhead is caused by moving all clauses from F_1 to F_2 to conclude the proof. This overhead increases with the number of parallel counters since the candidate invariant contains more clauses with more counters. However, the overhead is marginal in all cases and affects the runtime only slightly

In the next experiment, the candidate invariant is "true". As an invariant needs to be generated by the algorithm, these experiments require more time. For 10 parallel counters, EASY takes 69 seconds and NSMC 131 seconds. EASY is faster in these cases. But this speed up does not arise from the PDR-like

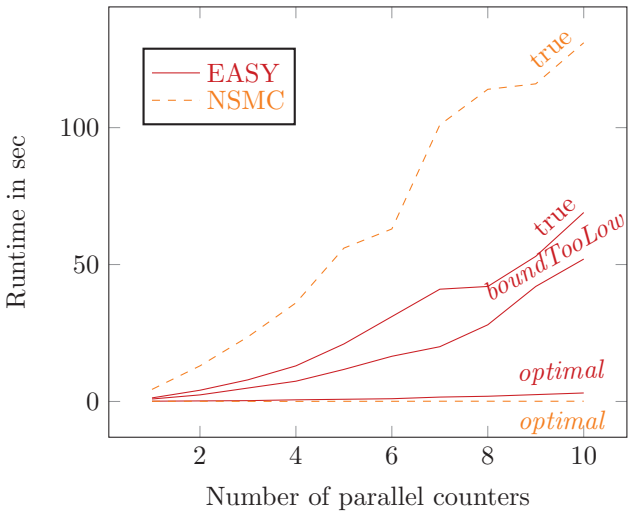


Figure 3.5: Runtimes of NSMC and EASY on the counter models

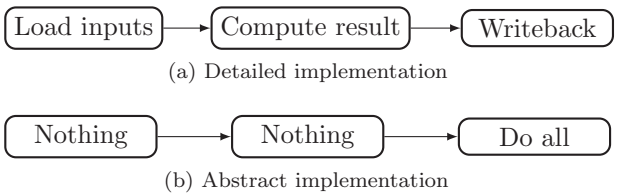
implementation but from the different implementations to check for intervals. In this example, the binary search of EASY is superior to the implementation of NSMC. For one parallel counter, EASY needs 134 of 165 of its calls to CBMC to find the optimal interval while NSMC requires 253 of its 260 calls.

The final experiment shows a case where EASY is significantly better than NSMC. The candidate invariant *boundTooLow* is an underapproximation of the reachable states. Both algorithms handle this problem quite differently and NSMC times out in all cases. NSMC will detect a counterexample that does not fulfill the candidate invariant after executing a method. Then, it will try to remove all predecessors of that counterexample from the candidate invariant, but will detect, that the initial state is a predecessor. Thus, the algorithm will add the counterexample and all its successors to the candidate invariant. Adding these predecessors and successors almost one by one causes a timeout. EASY will check if the candidate invariant describes safe states only, which is the case. Then, it will propagate clauses from F_1 to F_2. The clauses $mod_i \leq 300$ will not be moved, as we can easily show that states that fulfill *boundTooLow* can reach states that do not fulfill these clauses. Afterwards, the correct upper limit will be learned due to unsafe states. After propagating all clauses from F_2 to F_3, equivalence is proven.

The experiments on the counter have shown an example where the runtime is feasible even for bad initial candidate invariants. While the PDR approach of EASY can cause a neglectable overhead, it allows us to decide equivalence in cases that NSMC cannot handle.

3.4.2 Arithmetic Unit

For the models of the *Arithmetic Logic Unit* (ALU), we described a pipeline of length 3. The ALU has 4 registers that can store 3-bit values. It can handle three operations: ADD, SUB and NOP. The operations ADD and SUB add or subtract the values of two input registers and store the result in a target register.



(a) Detailed implementation

(b) Abstract implementation

Figure 3.6: Sketch of an implemented processors

The operation NOP does nothing. The implementations provide getter-methods for all registers and a nextStep-method to load a new operation that is given as input and execute one step of the pipeline.

Like sketched in Figure 3.6, the more detailed implementation loads the values of the input registers in the first step and stores them in special registers *in1* and *in2* that do not exist in the abstract model. Afterwards, the result of the operation is computed in the second step and the result is stored in a register *result* which also does not exist in the abstract model. The operation uses the values that are stored in *in1* and *in2*. Finally, the content of *result* is written into the target register in the third step.

The abstract implementation executes an operation completely in its third step and acts as a queue. It does not load input values before or stores the result, but uses the values directly from the registers and writes them back immediately.

To ensure that these models behave equivalently, we block new operations that could lead to conflicts, i.e., use input or output registers that are currently used by other operations on the pipeline. When such an operation is used as input for nextStep, we put the operation NOP in the pipeline instead. Preventing conflicts is important as conflicts would cause different behavior in the two models due to their different implementation. Similarly, when an invalid operation is given, NOP is put in the pipeline instead.

An optimal candidate invariant for this example contains the following information:

1. Corresponding registers and pipeline operations need to be equal in both models.
2. All registers need to be 3-bit values, i.e., between 0 and 7.
3. All operations in the pipeline need to be valid.
4. There are no conflicts in the pipeline.
5. The registers *in1*, *in2*, and *result* in the detailed model need to be correct.

Splitting the described candidate invariant into clauses leads to 95 clauses. For the experiments, we modify the optimal candidate invariant *alu-optimal* by

1. Removing the equality of $1 \leq i \leq 4$ registers: *i-regNotEqual*
2. Removing the correctness of the result in the detailed model: *noResult*
3. Adding a faulty equality of an input of the second operation and the output of the third operation: *wrongPipeEqual*

Candidate Invariant	**NSMC**		**EASY**	
	Time [s]	#Calls [-]	Time [s]	#Calls [-]
alu-optimal	0.4	3	0.8	4
1-regNotEqual	10.0	62	12.4	50
2-regNotEqual	16.0	101	25.1	96
3-regNotEqual	31.0	175	33.7	142
4-regNotEqual	35.0	201	44.4	188
noResult	T/O	-	T/O	-
wrongPipeEqual	T/O	-	25.3	104

Table 3.1: Runtimes and CBMC calls on the processor models

The candidate invariant *alu-optimal* describes exactly all reachable states and therefore should support the equivalence check significantly. For the candidates *i-regNotEqual*, the algorithms need to detect the equality of the corresponding registers.

Removing the correctness of the result from the candidate invariant is a lot harder than removing equality of registers as the correctness of the result is very complex to describe. It depends on the operation in the third step as well as the values of the registers. The result needs to contain the sum or the difference of the values in the input registers of the operation and is therefore dependent on seven different variables: the four registers, the two input registers of the third operation, and the type of the third operation.

Table 3.1 shows the runtime and the number of CBMC calls of NSMC and EASY for the different candidate invariants.

Proving correctness for *alu-optimal* and the candidate invariants *i-regNotEqual* takes longer with EASY compared to NSMC due to overhead like moving the clauses from F_1 to F_2. Using a PDR-like implementation does not speed up the equivalence check in these cases, similar to the experiment with the optimal invariant and the counters. Even though EASY requires a smaller amount of CBMC calls for the candidate invariants *i-regNotEqual*, the total runtime is higher than NSMC as the specific checks like checking whether all clauses can be moved take more effort than the calls during generalization that are not done by EASY due to low level optimization. The larger number of initial and learned clauses compared to the example with the counters lead to an increased overhead.

On the other hand, the underapproximation *wrongPipeEqual* timed out when run by NSMC but is decided within 25.3 seconds by EASY as EASY can easily detect the wrong clause and will not propagate it.

However, both equivalence checkers cannot decide equivalence when more complicated parts of the candidate invariant are missing, like shown in the experiments with *noResult*. The currently used heuristics cannot learn this behavior but instead remove single assignments that cause unsafe states. As the number of those assignments is too large to remove them one by one, the algorithms time out.

The experiments with the simple processor models confirm the observation from Section 3.4.1. Both algorithms are able to handle the models with a good

hypothesis. The hypothesis does not need to be optimal, but a bad hypothesis leads to a timeout for both algorithms. While EASY can cause neglectable overhead, it can also decide equivalence for candidate invariants that NSMC cannot handle.

3.4.3 Processor

In the final experiments, we will show, that our algorithms are able to handle bigger examples as well as long as the candidate invariant is good. For these experiments, we use a model for a processor. The processor has 8 3-bit registers. In addition, the processor has a 192-bit memory, that is separated into 64 3-bit words, a program counter that refers to the next executed operation within the memory, and a zero flag that is set after an arithmetic operations computes 0 and reset if an arithmetic operations computes a result that is not 0.

Each operation is coded by four 3-bit words. The first word refers to one of eight possible operation types and the remaining three words are used to code the arguments of the operation.

The eight operation types are

1. NOP: Does nothing, the three argument words need to be 0
2. STORE: Store the content of a register in the memory, the first two argument words refer to the position in the memory and the third one refers to the register
3. LOAD: Loads a word from the memory into a register, like STORE, the first two argument words refer to the memory address and the third one to the register in which the word is stored
4. ADD: Adds the content of two registers and stores it in a register, the first two argument words refer to the summands and the sum is stored in the register referenced by the third argument word
5. SUB: Subtracts the content of two registers and stores it in a register, the first argument word refers to the minuend, the second argument word to the subtrahend, and the difference is stored in the register referenced by the third argument word
6. JUMP: Change the program pointer to the memory location that is referenced by the first two argument words, the third argument word must be 0
7. JUMPZ: Change the program pointer to the memory location that is referenced by the first two argument words if the zero flag is set, the third argument word must be 0
8. EXIT: Stop the execution of the program, the argument words need to be 0

Whenever an invalid operation is read, e.g., a NOP with an argument word that is not 0, the operation is interpreted as EXIT instead. Similarly, when the program pointer refers to an operation that is not completely within the

memory, e.g., position 63 that would need argument words that are not within the memory, the operation is considered as EXIT as well.

The models have getter methods to read the registers, a method loadMemory that resets the model and loads the given content into the memory, and a method nextStep, that computes the next timestep of the processor execution.

Again, we use one model with pipelining and one without. The model with pipelining has a pipeline with 4 steps. The levels are similar to the ones in Section 3.4.2. In addition, the operation is read within a first step. Afterwards, the inputs are loaded, the result is computed, and written back. When an arithmetic operation is affected by a write-read conflict, the result is not written back. Similarly, when a STORE operation would write into an operation that is currently in the pipeline, the value is not written.

When a JUMP or JUMPZ operation is executed, the content of the pipeline is flushed and replaced with NOPs.

To handle the pipeline, the model needs additional variables to store the operations within the pipeline, the loaded inputs, and the computed result. It also uses a flag that is set after an EXIT operation and stops the further execution.

In comparison, the model without pipeline just executes the operation at the current program pointer in one go and increases the programpointer afterwards. To remain equivalent to the model with pipelining, this model also blocks operations that would cause a write-read conflict for the pipeline or write into a loaded operation. Even though this is not needed for a model without pipelining, these adjustments are needed for equivalence between the models. In addition, the model without pipelining has a *wait* variable. When *wait* is not 0 and nextStep is called, *wait* is reduced by 1 and nothing else is done. This is needed for equivalent behavior when the pipelined model needs to fill its pipeline in the beginning of the execution or after a jump. Thus, *wait* is initially and after a jump set to 3.

An optimal candidate invariant for these models is complex, as it needs to describe all the relations between the different implementations. It needs to contain

1. equality of registers, memory, and zeroflag
2. correct intervals for all variables
3. the program pointers correspond to each other such that when an operation leaves the pipeline it is executed by the model without pipelining
4. If *wait* is not 0, the pipeline contains the corresponding number of NOPs
5. The operations within the pipeline correspond to the operations within the memory
6. The exit-flag of the pipeline model is only set if it should be
7. Loaded inputs and computed result within the pipelining model are correct

This candidate invariant consists of 399 clauses.

As the computer that was used for the previous experiments failed to execute the NSMC call due to memory overflow, this experiment was instead run on

a Dual-Core AMD Opteron Processor 2222 SE with 3 GHz and 64 GB main memory.

Using the described optimal candidate invariant, the decision that the models are equivalent took EASY 8017 seconds and used the full 64 GB of memory. Weakening the candidate invariant only slightly, e.g., by removing the equality of two corresponding registers, will produce a miter during the execution of the algorithm, that leads to a memory overflow during the CBMC call.

This experiment has shown, that our algorithms can handle more complex models but comes close to the limits of our underlying model checker and requires an optimal candidate invariant.

In summary of the experiments, EASY has additional overhead that is not needed in some cases and equivalence can be decided faster by the "lighter" NSMC. However, EASY can decide equivalence in cases that NSMC cannot decide within the time limit. This is especially true when underapproximations are used as candidate invariants, that usually lead to a timeout in NSMC.

3.5 Conclusion

In this chapter, we presented two algorithms to prove functional equivalence of two hardware description on the system level. The presented algorithm uses a hypothesis that is stepwisely refined to approximate the set of all equivalent states of the two designs. The hypothesis allows to use the expert knowledge of a designer to speed up verification. Preliminary experimental results for two case studies, a scale parallel counter and a processor model, show that the runtime can be significantly reduced, even for complex designs, when the "right" hypothesis has been chosen.

In this chapter, we presented NSMC and EASY, two algorithms for functional equivalence checking of ESL description written in C++. The algorithms generate an inductive invariant to prove equivalence or detects a reachable counterexample that disproves equivalence. While NSMC uses a candidate invariant and advances it with each discovered spurious counterexample, EASY uses a PDR-like approach that can easily drop some clauses. If the models are not equivalent, the returned counterexample can be used as a starting point for debugging and if they are equivalent the algorithm returns an inductive invariant that can support future equivalence checks. We proposed an implementation of NSMC and EASY on top of the standard bounded model checker CBMC and presented experiments with three examples to show the applicability of the approach.

Chapter 4

Robustness Checking

While new technologies facilitate the creation of more advanced systems, the systems become more susceptible to transient faults. External factors like cosmic radiation may induce glitches in the system, which can lead to erroneous behavior. A circuit needs to be analyzed to ensure that no erroneous output is produced under transient faults, i.e., that the circuit is robust. Otherwise vulnerable gates have to be determined. The effects of transient faults may be masked due to logic, timing or electrical effects. During the analysis, variation in the gates' parameters must be taken into account.

An easy way to get a basic idea of the robustness of a circuit are simulation and testing. But these cannot prove the absence of possible errors except for very small circuits.

Our formal approaches are the first to analyze a *Single Event Transient* (SET) under logic, electrical, and timing masking including variation while considering all possible input assignments. Moreover, the analysis is conservative, i.e., if our approach decides that an SET may not cause an error this decision is safe under the given constraints for variation. Technically, we model the behavior using three-valued logic (0,1,X) where unknown values (X) conservatively approximate variation effects. The decision engine is based on *Boolean Satisfiability* (SAT). For brevity, we only consider combinational circuits. Along the lines of [19] the extension to sequential circuits is straight-forward.

The downside of a SAT formula that models the circuit in high detail can become very complex, especially if the signal that contains the SET reconverges. While the described monolithic approach can handle most circuits of the ISCAS-89 benchmarks, the number of variables to describe signals after a reconvergence increases exponentially with the depth after the reconvergence.

To prevent this complication, we also present a hybrid approach that combines simulation and formal verification to achieve scalability while keeping a detailed technology model. We describe a circuit in high detail dependent on the used technology. The resulting circuit is partitioned into a front and a back partition. Different partitionings are possible. For our work, we put all gates that are affected by reconvergence of the SET in the back partition. We can easily analyze the front partition by using a SAT solver. Afterwards, we simulate detected possible counterexamples on the whole circuit, generalize the counterexamples, and modify the SAT formula until a robustness can be decided.

As another advantage, the hybrid approach allows us to consider the delays in higher detail, as we can easily differentiate between the different delays of a gate depending on the input values during simulation.

In summary, the contributions of this chapter are two algorithms that

- consider logical, timing, and electrical masking,
- describe the gates in great detail, specific to the used technology and considering variability,
- allow a monolithic approach to decide robustness, where the whole circuit under an SET is described as SAT formula,
- allows a hybrid approach that uses SAT solving to check the front partition and uses simulation to verify detected counterexamples on the complete circuit, and
- can use composition as it partitions the circuit into two partitions to prevent reconvergence in the front partition.

The following Section 4.1 introduces some preliminaries. Sections 4.2 and 4.3 describe our monolithic and hybrid algorithms for robustness, respectively. Experiments with both algorithms are shown in Section 4.4 and Section 4.5 concludes this chapter.

4.1 Preliminaries

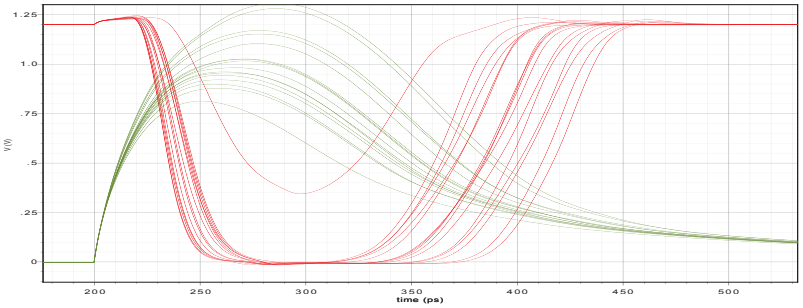
Three-valued logic extends Boolean logic by a value X, meaning that it is not known if that variable is 1 or 0. Operations on variables are extended accordingly, e.g., $1 \wedge X = X$ and $0 \wedge X = 0$. Three-valued logic allows us to describe uncertainties within the circuit conservatively by setting uncertain signals to X.

A *circuit* C consists of a set $C.G$ of gates, a set $C.I$ of input signals, and a set $C.O \subseteq C.G$ of outputs. To determine the connections between the gates, the functions $C.\mathsf{predecessors} : G \to 2^{I \cup G}$ and $C.\mathsf{successors} : (G \cup I) \to 2^{G}$ return the direct predecessors or successors of a gate or input signal.

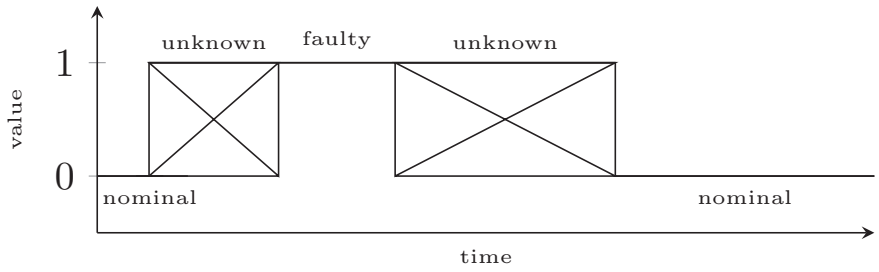
The set $C.SO \subseteq C.O$ describes safe outputs and contains all outputs that are secured, i.e., can correct the effects of an SET in this output, e.g., using Razor [15].

As we want to model an SET in our circuit, consider Figure 4.1a which shows SETs caused in a gate by an electron strike with constant energy but different variation of the gate. On the one hand, we can see that the change from the original value to the inverted value takes some time and the analog signal lies somewhere in between during this time. On the other hand, the time when the signal changes depends on the variability of the gate. As such, there is some uncertainty in between, as we are not sure how the circuit interprets the signal at these times, especially as they vary depending on the variability. To remain conservative, we model the SET in three-valued logic as shown in Figure 4.1b and consider the signal unknown during the described uncertainties.

An SET s that affects C is modeled by a number of parameters. The gate $s.g \in C.G$ describes the location where the SET strikes. The SET begins at



(a) Original Error Signals



(b) Model of an SET using three-valued logic

Figure 4.1: Modeling an SET

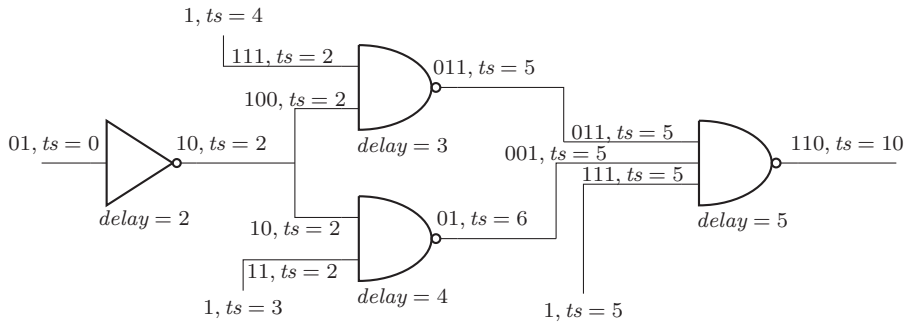


Figure 4.2: A part from the robust counter with WaveSAT

the time $s.b \in \mathbb{R}$ and ends at $s.e \in \mathbb{R}$. During an offset time at the beginning $s.o_b \in \mathbb{R}$ and at the end $s.o_e \in \mathbb{R}$, the signal becomes unknown. In between the offsets, the signal is inverted.

Each gate $g \in C.G$ is associated with an operation $g.\mathsf{op} : \{0,1,X\}^n \to \{0,1,X\}$ that describes the output of the gate under given input values. The function $g.\mathsf{delay} : \{0,1,X\}^n \times \{0,1,X\}^n \to \mathbb{R}$ returns the delay of the gate when the input changes, depending on the old and new input values. The values $g.d_{min}$ and $g.d_{max}$ denote the minimal and maximal delay in $g.\mathsf{delay}$, respectively.

For our algorithm, we only need to consider changes in the inputs where a single input changes to X or was previously X due to the form of the SET that always contains an X between two variables that are not X. When the signal of the SET is propagated, the time when a signal is X will become longer in any successor unless the SET is logically masked, due to our conservative handling of the delays. We take the minimal possible delay with the given input values when a signal changes to X and the maximal delay when it previously was X. By handling delays in this way, a single delay value is sufficient for each change from or to X without loosing any conservatism.

We introduce two models to describe the signals within the circuit. The first model uses *waveforms* similar to [51]. The models uses timesteps and thus, all functions that return time values, i.e., delays and the paramters of the SET, need to be natural numbers. This is no hard restriction as the size of a timestep can be scaled accordingly as these parameters are known beforehand. Each gate is associated with a waveform and a *timeshift* value. The waveform is a vector $(v_1, v_2, \ldots, v_l) \in Var^l$ that describes the changes of the output of the gate. The timeshift describes an offset from timestep 0 for the first variable in the waveform. The second variable models the following timestep and so on. The logical value before and after the waveform remain identical to the first and last variable, respectively. For example, a constant primary input i is described by timeshift 0 and a waveform that contains only one element. An example that describes a part of the counter circuit from Section 2.1 is given in Figure 4.2. In this example, the variables are given explicit values for better readability. The model itself uses variables that symbolically describe the behavior of the circuit and provide a SAT formula later on.

To calculate the waveform of a gate g, the waveforms of the predecessor gates are needed. All these waveforms need to be aligned to the same timeshift and have the same length. To change the input's waveforms to fulfill this condition,

padding is used. The minimal timeshift t_{min} among the inputs is determined. For an input with timeshift t, $t - t_{min}$ copies of the first variable are added in front of the waveform. This is feasible since the value does not change before t. The function $pad_{front} : (\mathbb{N} \times Var^*)^n \to (\mathbb{N} \times Var^*)^n$ performs this padding. The natural number describes the timeshift and the variables describe the waveforms. The value n is the number of predecessors of g and each waveform corresponds to one predecessor. In Figure 4.2 this can be seen as the output of the `not`-gate is padded with two 1 at the front to decrease the timeshift by 2 such that the two inputs of the `and`-gate have the same timeshift.

Afterwards, all waveforms are extended to the same length. Given the maximal length of the waveforms l_{max}, a waveform of length l is extended by adding $l_{max} - l$ copies of the last variable at the back to model the static value. This padding is executed by the function $pad_{back} : (\mathbb{N} \times Var^*)^n \to (\mathbb{N} \times Var^*)^n$. In our example, this affects the first input of the `and`-gate. As the second input has a length of 4, the first one needs to be padded accordingly and two 1 are added at the back of the waveform.

After modifying the inputs and ensuring the same timeshift and length, the waveform of g is determined. The waveform of g is as long as the padded waveforms of the predecessors. The i-th variable is defined by using the operation of the gate $op(g)$ with the i-th variable of each waveform of the predecessors. The timeshift of g is obtained by adding the delay of the gate to t_{min}[1]. The function $apply_{op} : V \times (\mathbb{N} \times Var^l)^n \to V \times \mathbb{N} \times Var^l$ applies the gate's operation to the waveform with

$$\begin{aligned} &apply_{op}(g, (t, (v_1^1, v_2^1, \ldots, v_l^1)), \ldots, (t, (v_1^n, v_2^n, \ldots, v_l^n))) = \\ &(g, t + delay_{min}(g), (op(g)(v_1^1, \ldots v_1^n), \ldots, op(g)(v_l^1, \ldots, v_l^n)) \end{aligned}$$

In Figure 4.2 this can be seen at both gates. At the `and`-gate, the `and`-operation is applied pairwise to the inputs and the delay of the gate is added to the timeshift of the inputs.

The complete process of padding and computing the new waveform is summarized in the function $wave : V \times (\mathbb{N} \times Var^*)^n \to V \times \mathbb{N} \times Var^*$ where

$$\begin{aligned} wave(g, (t^1, (v_1^1, v_2^1, \ldots, v_{l_1}^1)), \ldots, (t^n, (v_1^n, v_2^n, \ldots, v_{l_n}^n))) = \\ apply_{op}(g, pad_{back}(pad_{front}((t^1, (v_1^1, v_2^1, \ldots, v_{l_1}^1)), \ldots, \\ (t^n, (v_1^n, v_2^n, \ldots, v_{l_n}^n))))) \end{aligned}$$

In our implementation of [51], we reuse the same variable for two timesteps t and $t + 1$ in a waveform if the input variables of the considered gate are equal at both timesteps.

While this model can be used to describe timing analogously by choosing an according size for the timesteps, when the circuit contains a high number of gates with individual delays, an optimal stepsize that can exactly describe the timing behavior is very small and leads to very long waveforms. These waveforms will take a high effort to be handled and even with the reuse of variables lead to a reduced performance of the algorithm.

[1] Here, we use the model of [51] with a fixed delay. The maximal delay is taken into account in our approach in Section 4.2.

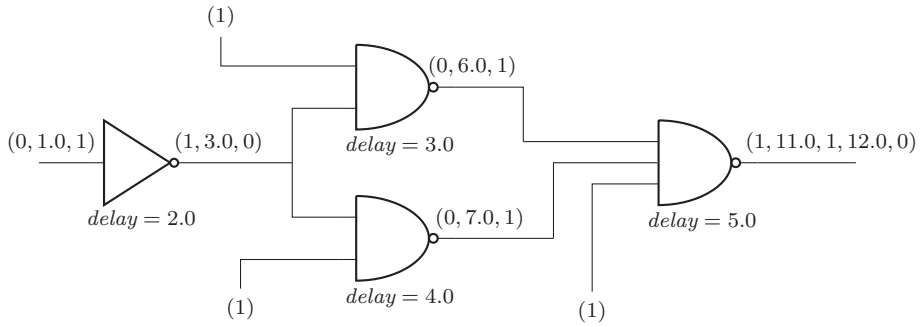


Figure 4.3: A part from the robust counter with the second model

With these observations in mind, another model for the signals was developed to avoid the described problems. In our second model, the output values of the gate over time are similarly described by a vector $g.wave \in Var^*$ where the set Var contains three valued variables. The vector $g.switch \in \mathbb{R}^{g.wave.\,\mathsf{size}()-1}$ contains the times when the output of the gate changes to the next variable. The first variable in $g.wave$ describes the signal before the SET affects g and the last variable describes the signal after the effects of the SET have passed g. The variables in between describe value changes at the output of g due to the SET. The time $g.switch_i$ defines when the signal changes from $g.wave_i$ to $wave_{i+1}$. For better readability, we combine the vectors $g.wave = (v_1, v_2, \ldots, v_n)$ and $g.switch = (t_1, t_2, \ldots, t_{n-1})$ to a single vector $g.signal = (v_1, t_1, v_2, t_2, \ldots, v_{n-1}, t_{n-1}, v_n)$. Thus, the signal of gate g is described by three vectors:

1. $g.wave = (v_1, v_2, \ldots, v_n)$ describes the different values the gate outputs.
2. $g.switch = (t_1, t_2, \ldots, t_{n-1})$ describes the times when the output changes.
3. $g.signal = (v_1, t_1, v_2, t_2, \ldots, v_{n-1}, t_{n-1}, v_n)$ combines $g.wave$ and $g.switch$ into a single vector.

For example, a constant signal would only require a single variable and no switch times and a gate g with $g.wave = (v_1, v_2)$ and $g.switch = (5)$ would change its output at time 5 from v_1 to v_2. The according vector $g.signal$ would be $(v_1, 5, v_2)$. While this model also uses a waveform to describe the output values of the gate, the switch times allow a finer scaling, as we do not consider individual timesteps but just use one element of the vector for any duration that the signal does not change.

In Figure 4.3, this model is used to describe the circuit from Figure 4.2. Like in Figure 4.2, the explicit values are merely for a better understanding as both models use variables without needing explicit values.

When determining the wave $g.wave$ and the switch times $g.switch$ of a gate $g \in C.G$, the signals of the predecessors $p_1, p_2, \ldots, p_n$ need to be computed already. We start by determining and sorting all existing switch times of the predecessors and store them in a vector *in-change*. We refer to the i-th element of a switch vector as s_i and define the length of $p_i.switch$ as l_i.

$$in\text{-}change = removeDouble(sort((p_1.s_1, \ldots, p_1.s_{l_1}, \ldots, p_n.s_1, \ldots, p_n.s_{l_n})))$$

The function *sort* sorts the elements in the vector and the function *removeDouble* removes values that are multiple times in the vector until only one such element is left. With these switch times for g, we consider all changes within the predecessor to consider them when computing the according variables. We can see the computation of the switch times of the `and`-gate in Figure 4.3. The first predecessor has one switch time of 1.0 and the second predecessor has a switch time of 3.0, resulting in the $\mathit{in\text{-}change} = (1.0, 3.0)$.

When we compute $g.\mathit{switch}$, we take every element from *in-change* and add the minimal delay $g.d_{min}$, resulting in

$$g.\mathit{switch} = (\mathit{in\text{-}change}_1 + g.d_{min}, \ldots, \mathit{in\text{-}change}_{\sum_{i=1}^{n} l_i} + g.d_{min})$$

The difference between the minimal and the maximal delay is considered in further steps explained in Section 4.3.

For the output of the `and`-gate, both switch times are considered and increased by 4.0, the delay of the `and`-gate. This results in the switch times 5.0 and 7.0 of the `and`-gate.

To determine the variables that describe the output of g, we start by introducing the function $\mathit{varAt} : C.G \times \mathbb{R} \to \mathit{Var}$. The variable $\mathit{varAt}(g', t)$ describes the output of g' at time t:

$$\begin{aligned}\mathit{varAt}(g', t) = g'.\mathit{wave}_i \text{ with } &(i = 1 \vee g'.\mathit{switch}_i < t)\\ &\wedge(i = g'.\mathit{switch}.\,\mathsf{size}() \vee g'.\mathit{switch}_{i+1} \geq t)\end{aligned}$$

The formula $(i = 1 \vee g'.\mathit{switch}_i \leq t)$ describes the last time switch before t. In the first possible case i equals 1 and refers to the first variable, which means there is no time switch before t. Otherwise, the switch time $g'.\mathit{switch}_i$ needs to be smaller or equal to t as it needs to describe the time before t. Similarly, the second formula $(i = g'.\mathit{switch}.\,\mathsf{size}() \vee g'.\mathit{switch}_{i+1} > t)$ describes that i either refers to the last variable or the following switch time needs to be after t.

Using the function *varAt*, we can easily define the wave of g as a vector with one element more than the switch time vector, i.e., a size of $g.\mathit{switch} + 1$.

$$g.\mathit{wave}_i = \begin{cases} g.\mathit{op}(\mathit{varAt}(p_1, t_i), \ldots, \mathit{varAt}(p_n, t_i)) & \text{for } i \neq g.\mathit{wave}.\,\mathsf{size}() \\ g.\mathit{op}(\mathit{varAt}(p_1, \tilde{t} + 1), \ldots, \mathit{varAt}(p_n, \tilde{t} + 1)) & \text{else} \end{cases}$$

$$\text{where } t_i = \mathit{in\text{-}change}_i \text{ and } \tilde{t} = \mathit{in\text{-}change}_{\sum_{j=1}^{n} l_j}$$

When using the function *varAt* exactly at a switch time, it returns the variable that describes the signal before the switch. We use this to describe all elements of $g.\mathit{wave}$ except for the last one. For the last one, we add 1 to the last switch time. This is obviously bigger than the last switch time and thus we get the last variable from the wave of every predecessor.

The second model needs to handle a smaller number of elements compared to the previous one even if every gate in the circuit has individual delays.

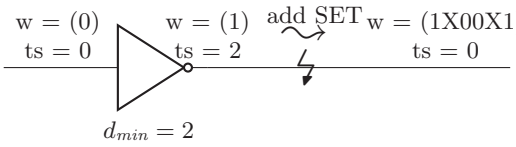


Figure 4.4: Generating an SET in a gate

4.2 Monolithic Robustness Checking

Our first approach checks the possibility that any output value can change from its nominal value during its sampling window due to a given SET. This algorithm decides robustness by using a monolithic approach, i.e., the whole circuit under the SET is formulated as SAT formula. A satisfactory assignment of the variables would describe a counterexample to robustness as the SAT formula requires an output signal that is not secured externally to behave erroneous.

If counterexamples exist, our algorithm returns one. Otherwise, the circuit is guaranteed to be robust under the given SET, i.e., the given SET cannot affect the output values for any assignment of input values.

To ensure conservativeness in our analysis we use X-values to model uncertainties. For example, transitions caused by the original SET in Figure 4.1a are approximated by X-values in the model of Figure 4.1b. For this approach, we use the first model introduced to describe the values of the signals, i.e., the model that uses timesteps.

Our algorithm decides the robustness in three steps:

1. Define the waveform for every gate iteratively by using the function *propagate*
2. Compare the waveform of each output during its sampling window to the nominal value
3. Return a counterexample or "circuit is robust" if no counterexamples exist

The function *propagate* defines the waveform of a gate under the given inputs. The execution of *propagate* corresponds to the call of multiple functions: The function *wave* as introduced in the preliminaries describes the initial waveform of the gate. The function *varDelay* describes the variable delay of the gate due to variability and other factors. The SET is given by the function *addSET* and electrical masking is considered in the function *elecMask*. Thus, we define $propagate : C.G \times (\mathbb{N} \times Var^*)^* \rightarrow C.G \times \mathbb{N} \times Var^*$ as $propagate = elecMask \circ addSET \circ varDelay \circ wave$. The individual functions used to compute *propagate* are introduced in the following sections.

4.2.1 Generation

The function $addSET : C.G \times \mathbb{N} \times Var^* \rightarrow C.G \times \mathbb{N} \times Var^*$ induces the effects of the SET s in the affected gate $s.g$. The inputs of *addSET* contain the current gate, the waveform of the gate and its timeshift. If the gate equals $s.g$, the SET is inserted into the waveform. Otherwise the inputs of this function are identical to its outputs.

In a first step, the algorithm ensures that the nominal behavior of the circuit before and after the SET is modeled within the waveform. Padding extends the waveform to start at timestep $s.b-1$ and to end at $s.e+1$ by applying the function $pad_{\text{SET}} : C.G \times \mathbb{N} \times Var^* \to C.G \times \mathbb{N} \times Var^*$ with

$$pad_{\text{SET}}(g,t,(v_1,\ldots,v_l)) = (g, min(t, s.b-1), (\underbrace{v_1,\ldots,v_1}_{s.b-t+1}, v_1,\ldots,v_l, \underbrace{v_l,\ldots,v_l}_{s.e-t-l+2}))$$

where the function *min* returns the minimum value among the inputs. If $s.b-t+1$ or $s.e-t-l+2$ are less than 0, no variables are added at the corresponding location.

In a next step, the SET as seen in Figure 4.1b is inserted. The values in the offset are replaced with c_X which is set to X and the variables in between are replaced with the negation of the variable at that location. The outputs of *addSET* are the gate, the new waveform, and timeshift.

Inserting the SET is done by the function $apply_{\text{SET}} : C.G \times \mathbb{N} \times Var^l \to C.G \times \mathbb{N} \times Var^l$ with

$$\begin{aligned} &apply_{\text{SET}}(g,t,(v_1,\ldots,v_l)) = \\ &(g,t,(v_1,\ldots,v_{s_{\text{SET}}-t}, \underbrace{c_X,\ldots,c_X}_{s.o_b \text{ times}}, \\ &\neg v_{s.b+s.o_b-t+1},\ldots,\neg v_{s.e-s.o_e-t+1}, \\ &\underbrace{c_X,\ldots,c_X}_{s.o_e \text{ times}}, v_{s.e-t+2},\ldots,v_l)) \end{aligned}$$

With these functions, *addSET* is defined as:

$$addSET(g,t,(v_1,\ldots,v_l)) = \begin{cases} apply_{\text{SET}}(pad_{\text{SET}}(g,t,(v_1,\ldots,v_l))) & \text{if } g = s.g \\ (g,t,(v_1,\ldots,v_l)) & \text{otherwise} \end{cases}$$

Example 3. Let us consider the counter in Figure 4.2. Note that for our examples we use explicit values for better understandability, even though our algorithm considers the variables symbolically. Let the primary input be a constant 0, which corresponds to the waveform (0) and the timeshift 0. Let the SET be $(g_{not}, 1, 4, 1, 1)$, where g_{not} is the `not`-gate in the circuit. This means the SET strikes in g_{not} at timestep 1 and lasts until timestep 4. The offset during which the value of the signal becomes unknown is 1 at the front and the end. The insertion of the SET is shown in Figure 4.4. Before the SET is inserted, the regular waveform of g_{not} is computed. The resulting waveform is (1) with timeshift 2. Afterwards, the waveform needs to be padded to include the timestep before and the timestep after the SET. The SET starts at timestep 1 and ends at timestep 4. Therefore the waveform needs to include the timesteps 0 and 5. The padded waveform is (111111) with timeshift 0. Inserting the SET sets the variables at the beginning and the end of the SET to X. The variables in between are negated. When the SET is inserted, the waveform changes to (1X00X1).

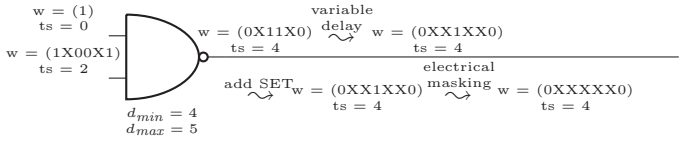


Figure 4.5: Propagating a signal considering logical, timing, and electrical masking as well as variability

4.2.2 Propagation

This section describes the propagation of values in the circuit by introducing the waveforms for each gate and defining the relation between the variables. The propagation considers the variable delays of the gates as well as electrical masking.

The real delay of a gate at a certain time depends on different factors like hardware variability, the input signals, or external influences. To approximate this behavior without modeling all details that affect the delay of a gate, we define a minimal and a maximal delay for each gate. If the output value of a gate at a certain timestep differs for different possible delays, the output value becomes unknown.

Let $\mathit{diff} = g.d_{max} - g.d_{min}$ be the difference between the minimal and the maximal delay of g. After generating the initial waveform, it is padded in the back for *diff* variables to consider the latest possible output as well. This is realized by the function $pad_{delay} : C.G \times \mathbb{N} \times \mathit{Var}^* \to C.G \times \mathbb{N} \times \mathit{Var}^*$.

Example 4. Let us return to our example from Figure 4.2. In this example, we consider the lower `nand`-gate that follows the `not`-gate from Example 3. The gate is pictured in Figure 4.5. The `nand`-gate g_{nand} has two predecessors with the waveforms (1) from a `not`-gate that is not affected by the SET and (0X11X0) from the `not`-gate from Example 3. The resulting initial waveform of g_{nand} is (0X11X0) with timeshift 4. Next, the variable delay is considered. The difference *diff* between the maximal and the minimal delay is $\mathit{diff} = d_{max} - d_{min} = 5 - 4 = 1$. Therefore, we use padding to add one variable at the back, which results in the waveform (0X11X00).

For the variable delay, each variable in the waveform is compared to the *diff* previous variables. If the values are equal, the output at that time is identical for all applicable delays and remains the same. If the compared values are different, the resulting output is set to X as different delays within the valid range cause different output values. The function $apply_{delay} : C.G \times \mathbb{N} \times \mathit{Var}^l \to C.G \times \mathbb{N} \times \mathit{Var}^l$ with $apply_{delay}(g, t, (v_1, \ldots, v_l)) = (g, t, (v'_1, \ldots, v'_l))$ describes this step. In this function, the new variables v'_i are defined as

$$v'_i = \begin{cases} v_i & \text{if } v_{i-\mathit{diff}} = \cdots = v_i \\ X & \text{otherwise} \end{cases}$$

The complete process of adding the variable delay is summarized in the function $\mathit{varDelay} : C.G \times \mathbb{N} \times \mathit{Var}^* \to C.G \times \mathbb{N} \times \mathit{Var}^*$ with $\mathit{varDelay} = apply_{delay} \circ pad_{delay}$.

Example 5. In our example from Figure 4.5, we compare the value of each variable to the value of the previous variable because $\mathit{diff} = 1$. If the values are

equal, the variable is not changed, otherwise it is replaced with X. The first variable has no previous variable and remains unchanged. The third variable needs to be replaced with X because its value is 1 and the value of the previous variable is X. The fourth variable remains unchanged, because the value of the third and fourth variable are both 1. The resulting waveform is (0XX1XX0).

In a next step, the earlier described function *addSET* is used to induce the SET, if $g = s.g$. In our example no SET is induced, as $g_{nand} \neq s.g$. Afterwards, electrical masking is applied. The application of electrical masking is done by using the function $elecMask : C.G \times \mathbb{N} \times Var^l \rightarrow C.G \times \mathbb{N} \times Var^l$ presented in Section 4.2.3.

Some low level optimizations were implemented to improve our algorithm by efficiently reducing the amount of used variables.

When a waveform consists of a single variable, the according signal is constant. If inputs of a gate g are constant, the output of g is also constant. We use a waveform with a single variable for g. The single variable depends on the input values and the operation of g. Due to the output of g being constant, the timeshift is not relevant to define the output of g. The waveform with a single variable can be padded towards any timestep if needed. Furthermore, variable delays and electrical masking do not need to be considered for constant signals.

Within the fanout of the SET, waveforms usually consist of five blocks of variables as long as the effects of the SET do not split and reconvergence. Either all of these variables will have the same value or the values will still correspond to the SET and have the form $vX\neg vXv$. When computing the variable delay, whenever different variables are compared, we use a variable from the second or fourth block instead of introducing additional variables. If the values of all variables are equal, it does not matter which variable is picked, and if the second and fourth block's value is X, the compared variables have different values and the output of X in the modified waveform is correct.

4.2.3 Electrical Masking

Electrical properties of the gates mask short glitches. A glitch is a change of a signal that lasts for a finite time and switches back to its original value afterwards.

Let the threshold t be the maximal duration of glitches masked by gate g. Every glitch shorter than or equal to t is removed by electrical masking. As simplification, we set t to half the minimal delay of g. Let us assume, there are two variables v_1 and v_2 with the same value *val* on the waveform and there are t or less variables between them. In this case, the variables between v_1 and v_2 need to be set to the value *val* to remove the glitch. If multiple glitches exist in the waveform, our process starts at the front and the processing of an earlier glitch can remove a later glitch.

To prepare the decision of electrical masking on the waveform $(v_1, \ldots, v_l)$, we introduce three vectors $\vec{v}^{\,is0} = (v_1^{is0}, \ldots v_l^{is0})$, $\vec{v}^{\,is1} = (v_1^{is1}, \ldots v_l^{is1})$, and $\vec{v}^{\,isX} = (v_1^{isX}, \ldots v_l^{isX})$. A variable v_i^{is0} is 1 iff v_i is equal to 0. The variables v_i^{is1} and v_i^{isX} are defined likewise for 1 and X, respectively.

Example 6. Let us apply this step to our example from Figure 4.5. The current waveform from g_{nand} is (0XX1XX0). Since exactly the first and the

last variable are equal to 0, the vector $\vec{v}^{\,is0} = (1000001)$. The other vectors are $\vec{v}^{\,is1} = (0001000)$ and $\vec{v}^{\,isX} = (0110110)$.

The following explanation describes how electrical masking towards 0 is handled. These operations are executed equivalently for 1 and X.

After calculating the vectors $\vec{v}^{\,is0}$, $\vec{v}^{\,is1}$, and $\vec{v}^{\,isX}$, we check for each variable v_i on the waveform, if it could be changed to 0 due to electrical masking. If v_j and v_k are the closest variables to v_i that are equal to 0 and have a distance of $k - j \leq t$ variables, v_i could be changed to 0.

For every variable $v_{i-t}, \ldots, v_{i-1}$, it is checked if that variable is the last variable before v_i that is equal to 0. A variable v_j is such a variable iff it is equal to 0 and all variables between v_j and v_i, i.e., $v_{j+1}, \ldots, v_{i-1}$, are not equal to 0. For this comparison, we use the prepared variables $\vec{v}^{\,is0}$:

$$v_j^{last\text{-}0} = v_j^{is0} \wedge \neg v_{j+1}^{is0} \wedge \cdots \wedge \neg v_{i-1}^{is0}$$

We check the t variables behind v_i similarly for the first variable after v_i that is equal to 0. For v_j after v_i the variable $v_j^{first\text{-}0}$ is defined:

$$v_j^{first\text{-}0} = v_j^{is0} \wedge \neg v_{j-1}^{is0} \wedge \cdots \wedge \neg v_{i+1}^{is0}$$

Example 7. The threshold for glitches t is half the minimal delay of g, i.e., in our example $t = \frac{g_{nand}.d_{min}}{2} = 2$. For our example, we consider the fourth variable v_4. The single 1 is a glitch that will be removed and replaced by X. Since $t = 2$, we need to consider the two variables before and after v_4. Let us only consider electrical masking towards X. We need to compute the variables $v_2^{last\text{-}X}$, $v_3^{last\text{-}X}$, $v_5^{last\text{-}X}$, and $v_6^{last\text{-}X}$. The variable $v_2^{last\text{-}X}$ is 0 because there is another X between v_2 and v_4, i.e., v_3. Since v_3 is equal to X and there are no further variables between v_3 and v_4, $v_3^{last\text{-}X} = 1$. Similarly, $v_5^{last\text{-}X} = 1$ and $v_6^{last\text{-}X} = 0$ hold.

After determining the location of the closest variables to v_i that are 0, we can decide if it is possible, that v_i is masked towards 0. If any two variables $v_j^{last\text{-}0}$ and $v_k^{last\text{-}0}$ are equal to 1 and the difference between j and k is t or less, v_i could be changed to 0, which is presented by the variable $v_i^{potential\text{-}0}$:

$$v_i^{potential\text{-}0} = \bigvee_{j \in \{i-t,\ldots,i-1\}} \; \bigvee_{k \in \{i+1,\ldots,j+t+1\}} v_j^{last\text{-}0} \wedge v_k^{first\text{-}0}$$

Example 8. In our example, v_4 will be masked towards X, so we will compute $v_4^{potential\text{-}X}$:

$$\begin{aligned} v_4^{potential\text{-}X} &= \bigvee_{j \in \{4-2,\ldots,4-1\}} \Big(\bigvee_{k \in \{4+1,\ldots,j+2+1\}} (v_j^{last\text{-}X} \wedge v_k^{first\text{-}X}) \Big) \\ &= (v_2^{last\text{-}X} \wedge v_5^{first\text{-}X}) \vee (v_3^{last\text{-}X} \wedge v_5^{first\text{-}X}) \vee (v_3^{last\text{-}X} \wedge v_6^{first\text{-}X}) \\ &= 1 \end{aligned}$$

In case v_i could be changed into more than one value, we change v_i according to the earlier variable before v_i.

Deciding which potential change is executed is realized by checking all possible combinations of variables:

$$\begin{aligned} v_i^{\textit{change-0}} = v_i^{\textit{potential-0}} \wedge \\ (\neg v_i^{\textit{potential-1}} \vee \bigvee_{j\in\{i-t,\ldots,i-1\}} (v_j^{\textit{last-0}} \wedge \neg \bigvee_{k\in\{i-t,\ldots,j-1\}} v_k^{\textit{first-1}})) \wedge \\ (\neg v_i^{\textit{potential-X}} \vee \bigvee_{j\in\{i-t,\ldots,i-1\}} (v_j^{\textit{last-0}} \wedge \neg \bigvee_{k\in\{i-t,\ldots,j-1\}} v_k^{\textit{first-X}})) \end{aligned}$$

Example 9. In the example, it can be shown that $v_4^{\textit{potential-0}} = v_4^{\textit{potential-1}} = 0$. This leads to the conclusion, that $v_4^{\textit{change-X}} = 1$. For every other variable than v_4, electrical masking will not change the value. The resulting waveform for g is (0XXXXX0).

The function $\textit{elecMask} : C.G \times \mathbb{N} \times \textit{Var}^l \to C.G \times \mathbb{N} \times \textit{Var}^l$ summarizes the electrical masking with $\textit{elecMask}(g, t, (v_1, \ldots, v_l)) = (g, t, (v'_1, \ldots, v'_l))$ where

$$v'_i = \begin{cases} \textit{val} & \text{if } v_i^{\textit{change-val}} = 1,\ \textit{val} \in \{0, 1, X\} \\ v_i & \text{otherwise} \end{cases}$$

When two variables on the waveform next to each other are equal, the resulting variables from electrical masking will be equal as well. In those cases, we can reuse the variable that describes electrical masking in the previous timestep.

Additionally, we check the length of equal variables in a row before considering electrical masking. If this variable block is longer than t, electrical masking within that block is impossible and is not checked.

4.2.4 Observation of Erroneous Behavior

By executing the described steps for each gate, it is possible to represent the whole circuit in form of a SAT formula using three-valued logic. This formula is used to check if erroneous output in the sampling window is possible.

For the observation of an error, the nominal value of each gate is computed. If any output differs from the nominal output in the given sampling window, an error occurs. This check is realized by the function $\textit{gate-error} : G.O \times \mathbb{N} \times \textit{Var}^* \to \textit{Var}$ with

$$\textit{gate-error}(g, t, (v_1, \ldots, v_l)) = g \notin G.SO \wedge \bigvee_{1\le i\le l} (v_i \oplus \textit{output}_{nom}(g, in_1, \ldots, in_{|I|})$$

where the variable in_i corresponds to the i-th input.

The given SET can possibly lead to erroneous behavior if at least one of these checks returns 1. This is checked by `or`-operations over all these checks.

The final variable *overall-error* describes, if an error occurs under the given SET:

$$\textit{overall-error} = \bigvee_{g\in C.O} \textit{gate-error}(g, t_g, \vec{v}_g)$$

The variable t_g describes the timeshift of an output gate g and $\vec{v}_g$ is its waveform.

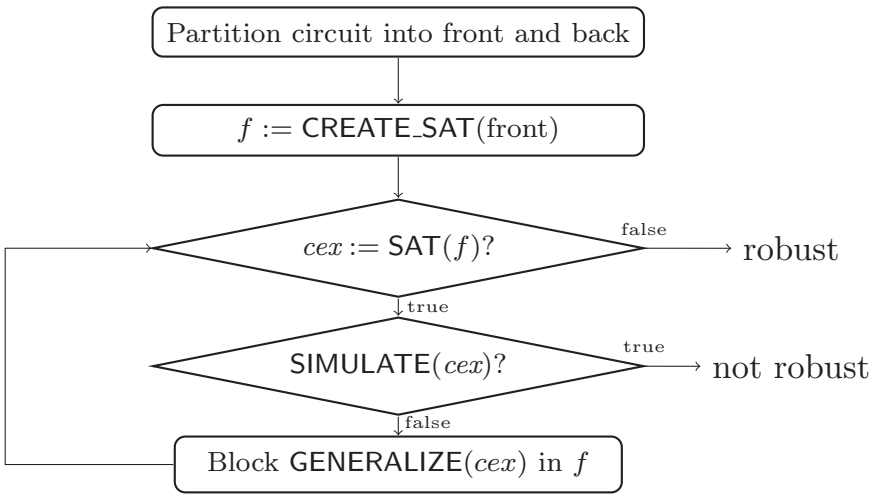


Figure 4.6: Sketch of the algorithm

To find a counterexample, the variable *overall-error* is set to 1. If the resulting SAT formula is satisfiable, the solution corresponds to a counterexample. Otherwise, the circuit is robust against the given SET since no assignment of variables exists that can lead to erroneous output. This final decision is done by a SAT solver that is used to find a solution for the SAT formula or proof its unsatisfiability.

4.3 Hybrid Robustness Checking

The interface of the hybrid approach is identical to the monolithic one and decides if a given circuit C is robust against a given SET s. If C is not robust, a counterexample to the robustness is returned. A counterexample contains an input assignment that leads to faulty behavior of C under s.

This approach partitions the circuit into two parts to prevent the high complexity that the monolithic approach needs when handling the splitting and reconverging of the SET. As the SAT formula only describes the behavior of the front part, the complexity is significantly lower than the corresponding formula of the monolithic approach. In return, during the hybrid approach each generated counterexample needs to be simulated. If a counterexample is spurious, it is used to refine the SAT formula. Thus, a usual run of the hybrid approach is a back and forth between SAT solving and simulation until a decision is made. To efficiently handle the different delays of each gate, the hybrid approach uses the second model to describe the signal, i.e., the model with a vector of switch times.

The algorithm is sketched in Figure 4.6. In the first step, the circuit is partitioned into a front and back partition. The partitioning is sketched in Figure 4.7. Afterwards, a SAT formula f is created to describe the behavior of the front partition under s. A satisfying assignment of f is an assignment to the primary inputs under which the effects of the SET can reach the back area. If f is not satisfiable, the circuit is guaranteed to be robust against s. Otherwise, the detected assignment is a counterexample cex against robustness

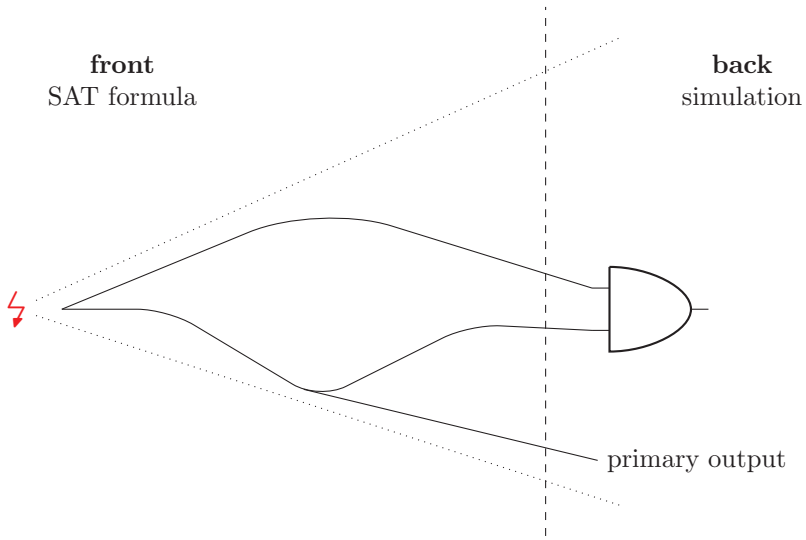


Figure 4.7: Partition in front and back

of the front partition. The assignment *cex* is simulated on the whole circuit. If the primary outputs of C are affected by s under the assignment of *cex*, the circuit is not robust and we call *cex* a *real counterexample*. Otherwise, *cex* is a *spurious counterexample*, that is generalized and blocked in f. Afterwards, we continue to check if the modified f is satisfiable until we can make a decision if the circuit is robust.

In the following sections, we will explain the proposed algorithm in detail. We start with the top level algorithm in Section 4.3.1 and describe the underlying algorithms in the following sections.

4.3.1 The Algorithm ROBUST_CHECK

Algorithm 11 implements the sketch of Figure 4.6. In the beginning we partition the circuit into front and back partion. For the used partitioning, we want all gates in which the SET reconverges to be in the back partition. This partitioning leads to easy SAT formulas that can quickly be solved, To determine the gates in the back partition we use an approach similar to breadth first search towards the outputs starting in the gate $s.g$ in lines 1 – 12. We also prepare the set O_{front} that contains all primary outputs as well as all gates in the front partition that have successors in G_{back} in line 13.

After the circuit is partitioned into the front and back partition, we create a SAT formula f to model the front partition. This SAT formula is satisfiable if the SET can reach the back partition and there is a possible fault in the circuit. Line 14 calls the respective algorithm CREATE_SAT.

While f is satisfiable, potential counterexamples exist which show the SET reaches the back partition. If f is satisfiable, we get an assignment a in line 16. The input assignment a_{in} of a is simulated on the complete circuit by calling the algorithm SIMULATE in line 18 to check if the potential counterexample is real. The simulation is very similar to the generation of the SAT formula, but simulates delays for the given input values accurately using the delay maps of

Algorithm 11: ROBUST_CHECK

input : a circuit C and an SET s
output : an assignment that leads to faulty behavior in C under s or "robust" if no such assignment exists

```
1  G_found := ∅
2  G_back := ∅
3  Q_search :=< s.g >
4  while Q_search ≠<> do
5  |  g := Q_search.pop()
6  |  if g ∈ G_found then
7  |  |  G_back := G_back ∪ {g}
8  |  else
9  |  |  G_found := G_found ∪ {g}
10 |  end
11 |  foreach g' ∈ C.successors(g) do Q_search.push(g')
12 end
13 O_front := {g ∈ G \ G_back|(C.successors(g) ∩ G_back) ≠ ∅ ∨ g ∈ C.O}
14 f := CREATE_SAT(C, G \ G_back, O_front, s)
15 while SAT(f) do
16 |  a := getAssignment(f)
17 |  a_in := a_in : C.I → {0, 1, X} with a_in(i) = a(i)
18 |  (a_sim, real) := SIMULATE(a_in, C, s, C.G)
19 |  if real then
20 |  |  return a
21 |  else
22 |  |  foreach o ∈ {g ∈ C.O \ C.SO|f.po-faulty_g(a)} do
23 |  |  |  f.po-faulty_o.addClause(¬ GENERALIZE(o, a_sim, a, C, s))
24 |  |  end
25 |  end
26 end
27 return "robust"
```

gates. If the counterexample is real, it proves that the circuit is not robust and the corresponding assignment is returned in line 20.

When the counterexample is spurious and the SAT formula assumes for a non-safe primary output $o \in C.O \setminus C.SO$ that o is affected by the SET, i.e., $f.po\text{-}faulty_o$ is true, we determine a minimal assignment that prevents the SET from reaching o by calling GENERALIZE. We add the generalized assignment to $f.po\text{-}faulty_o$ which is meant to be true if the SET could reach o. Since the assignment prevents o from being affected by the SET, we can modify $f.gate\text{-}faulty_o$ accordingly in line 23.

The loop from lines 15 – 26 further modifies f until either a real counterexample is found or f is not satisfiable any more. In the later case, the loop terminates and the algorithm returns that C is robust in line 27.

4.3.2 The Algorithm CREATE_SAT

The algorithm to create the SAT formula that describes the front partition starts by initializing the SAT formula f with "true" in line 1 of Algorithm 12. We use a queue to iteratively compute the waveform and switch times for each gate. The queue Q is initialized with all successors of the primary inputs in lines 2 – 5. While Q is not empty, we pop the front element g of the queue. If g still has predecessors whose waveform is not defined yet, g is pushed to the back of Q as seen in lines 8 – 9.

Otherwise, we determine the initial signal $g.signal$ of g in line 11 by calling COMPUTE_WAVE. The signal is further modified by considering variable delays in line 12 by calling VARIABLE_DELAY, adding the SET in case $g = s.g$ in line 13 by calling ADD_SET, and finally considering electrical masking in line 14 by calling ELECTRICAL_MASKING.

After $g.signal$ computed, we add all successors of g to the queue that have not been added yet and are part of the front partition. This is done in lines 15 – 19.

After the loop is done, we require all inputs to be different from X in lines 22 – 24. Thus, we have only boolean inputs as all nominal behavior of the circuit is boolean as well and the value X can only be assigned due to the SET.

In a next step, we introduce a subformula $f.fo\text{-}faulty_{fo}$ of f for each non-safe front output $fo \in O_{front} \setminus C.SO$. This subformula evaluates to "true" or X if o is affected by the SET, i.e., faulty. The output fo is faulty iff the signal of fo is not constant. These subformulas are generated for each output in lines 25 – 28.

We initialize further subformulas $f.po\text{-}faulty_{po}$ for each non-safe primary output $po \in C.O \setminus C.SO$. The formula $f.po\text{-}faulty_{po}$ estimates conservatively if po is affected by the SET. Initially, the formula is true iff at least one front output in the fanin of po is faulty. Later on, $f.po\text{-}faulty_{po}$ will be modified to store the information about detected spurious counterexamples. The according loop is in the lines 29 – 32.

The final subformula *overall-faulty* introduced in line 33 is true iff at least one front output is faulty. The variable *overall-faulty* describes that there is a potential error in the circuit. As we require counterexamples that describe such faults, we require *overall-faulty* to be different from 0 by adding the according clause to f in line 34.

The resulting SAT formula describes the behavior of the front partition and is only satisfiable if there is a fault in the front partition that could reach a primary output. It is returned in line 35.

Algorithm 12: CREATE_SAT

input : a circuit C, a set G_{front} of gates without potential reconvergence of the SET, a set $O_{front} \subseteq G_{front}$ of output gates of the front part, and an SET s

output : a SAT formula that is satisfiable iff there can be a fault in the front output in our model of C under the SET s

```
1  f := true
2  Q :=<>
3  foreach g ∈ ⋃_{i∈C.I} C.succesors(i) do
4  |  Q.push(g)
5  end
6  while Q ≠<> do
7  |  g := Q.pop()
8  |  if ∃p ∈ C.predecessors(g) : p.wave = ⊥ then
9  |  |  Q.push(g)
10 |  else
11 |  |  COMPUTE_SIGNAL(g, C, f)
12 |  |  VARIABLE_DELAY(g)
13 |  |  if g = s.g then ADD_SET(g, s, f)
14 |  |  ELECTRICAL_MASKING(g)
15 |  |  foreach suc ∈ C.successors(g) do
16 |  |  |  if suc.wave = ⊥ ∧ ¬Q.contains(suc) ∧ suc ∈ G_front then
17 |  |  |  |  Q.push(suc)
18 |  |  |  end
19 |  |  end
20 |  end
21 end
22 foreach i ∈ C.I do
23 |  f.addClause(i ≠ X)
24 end
25 foreach fo ∈ O_front \ C.SO do
26 |  w = fo.wave
27 |  f.fo-faulty_fo := new SAT subformula of f:
   |        ¬(w_0 = w_1 ∧ ··· ∧ w_{n-1} = w_n)
28 end
29 foreach po ∈ C.O \ C.SO do
30 |  w = po.wave
31 |  f.po-faulty_po := new SAT subformula of f:
   |        ⋁_{fo∈fanin(po)} f.fo-faulty_fo
32 end
33 overall-faulty := ⋁_{o∈C.O\C.SO} f.po-faulty_o
34 f.addClause(overall-faulty ≠ 0)
35 return f
```

4.3.3 The Algorithms to Compute the Signals

During the execution of CREATE_SAT, the signals for each gate are computed. This is realized by using the algorithms COMPUTE_SIGNAL, VARIABLE_DELAY, ADD_SET, and ELECTRICAL_MASKING. The algorithm COMPUTE_SIGNAL computes the initial signal, VARIABLE_DELAY adds consideration of variable delays due to variability or different input values, ADD_SET inserts the SET s into the signal in case the gate under consideration is $s.g$, and ELECTRICAL_MASKING modifies the signal to consider electrical masking.

The Algorithm COMPUTE_SIGNAL

Algorithm 13: COMPUTE_SIGNAL

input : a gate $g \in C.G$, a circuit C, and a SAT formula f that is currently constructed

1 $n := |C.\,\mathsf{predecessors}(g)|$
2 $\{p_0, \ldots, p_n\} := C.\,\mathsf{predecessors}(g)$
3 $(i_0, \ldots, i_n) := (0, \ldots, 0)$
4 $\mathit{current\text{-}in} = (c_0, \ldots, c_n) := (p_0.\mathit{wave}_0, \ldots, p_n.\mathit{wave}_0)$
5 $w_g := (w_0)$
6 $s_g := ()$
7 $f.\,\mathsf{addClause}(w_0 = g.\,\mathsf{op}(\mathit{current\text{-}in}))$
8 **while** $\exists j \in \{0, \ldots, n\} : i_j < p_j.\mathit{switch}.\,\mathit{size}()$ **do**
9 $\quad m := \mathsf{min}(p_j.\mathit{switch}_{i_j} | j \in \{0, \ldots, n\})$
10 $\quad j := \mathsf{indexOf}(m)$
11 $\quad \mathit{current\text{-}in} := (c_0, \ldots c_{j-1}, p_j.\mathit{wave}_{i_j+1}, c_{j+1}, \ldots, c_n)$
12 $\quad i_j := i_j + 1$
13 $\quad s_g := s_g \circ (m + g.d_{min})$
14 $\quad v :=$ new Variable
15 $\quad w_g := w_g \circ (v)$
16 $\quad f.\,\mathsf{addClause}(v = g.\,\mathsf{op}(\mathit{current\text{-}in}))$
17 **end**
18 $g.\mathit{wave} := w_g$
19 $g.\mathit{switch} := s_g$

The algorithm COMPUTE_SIGNAL determines the waveform and switch times of a gate g depending on the inputs and gates' operation and is shown in Algorithm 13.

We define an index for each predecessor of g that refers to a position in the waveform of the predecessor. The current indices refer to the current inputs and will increase while the algorithm moves forward in time. We also introduce the current inputs *current-in* that depend on the current indexes. As a final preparation, we define the first variable of $g.\mathit{wave}$. The preparations are done in lines 1 – 7.

While there is still an index that refers to an existing switch time, we determine the minimal switch time m and the corresponding index j in the lines 9 and 10. We adjust the current inputs by using the next variable of the j-th input in line 11 and increase the index i_j by one in line 12. We add the next

switch time which is the determined minimal switch time and the added minimal delay of g, i.e., $m + g.d_{min}$ in line 13 and a new variable v to the waveform which needs to be equal to the output of g with the changed inputs in the lines 14 – 16.

The Algorithm VARIABLE_DELAY

Algorithm 14: VARIABLE_DELAY

```
   input  : a gate g ∈ C.G
1  (t_1, ..., t_n) := g.switch
2  for j := 2, 4, ..., n do
3  |  t_j := t_j + (g.d_max − g.d_min)
4  end
5  g.switch := (t_1, ..., t_n)
```

When considering the variable delay of g in the algorithm VARIABLE_DELAY shown in Algorithm 14, we exploit that there is no reconvergence in g as g is in the front partition. This leaves three cases for the waveform:

1. The output is constant
2. The output has the form of the SET: $vX\neg vXv$
3. The output has the form of the SET with the middle part removed: vXv

As the variables do not have assigned values at this time, it is impossible to decide which case will hold, however we can do the following modification in all cases. Since we will only modify the switch times in this algorithm, the semantics of the output will not change if it is constant. Otherwise, we hold the output at X as long as possible within the limits of the delays to remain conservative. Since in a non-constant output every second variable is X, we set the switch times at those locations to the maximum delay instead of the minimum delay. Therefore we use the minimal delay when we change the output to X and use the maximal delay when we change back to another value.

The Algorithm ADD_SET

Algorithm 15: ADD_SET

```
   input  : a gate g ∈ C.G, an SET s, and a SAT formula f
1  //Signal of g is constant before SET is induced
2  v := g.wave_0
3  v_X, v_N := new Variable
4  g.wave := (v, v_X, v_N, v_X, v)
5  g.switch := (s.b, s.b + s.o_b, s.e − s.o_e, s.e)
6  f.addClause(v_X = X ∧ v_N = ¬v)
```

If we induce the SET s into a gate g, we use the algorithm ADD_SET shown in Algorithm 15. The waveform of g needs to be constant and only contain one variable as only the SET leads to a change in the output of a gate. In line 4 we

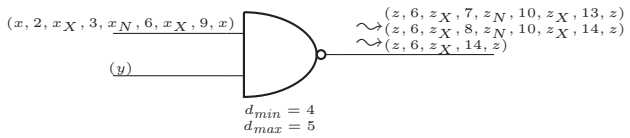


Figure 4.8: Example for generation of waveform and switch times of a gate considering variable delays and electrical masking depending on gate

generate a new waveform for g which corresponds to the SET and in line 5 we set the switch times according to the parameters of the SET.

The Algorithm ELECTRICAL_MASKING

Algorithm 16: ELECTRICAL_MASKING

```
   input  : a gate g ∈ C.G
1  t := g.d_min / 2
2  for j := 0, ..., g.switch.size() − 2 do
3  |  for k := j + 1, ..., g.switch.size() − 1 do
4  |  |  //Equal checks between variables check if the variables are equal,
   |  |  not their values
5  |  |  if g.switch_k − g.switch_j ≤ t ∧ g.wave_j = g.wave_{k+1} then
6  |  |  |  g.wave := (g.wave_0, ..., g.wave_{j−1},
   |  |  |                g.wave_{k+1}, ..., g.wave_{g.wave.size()−1})
7  |  |  |  g.switch := (g.switch_0, ..., g.switch_{j−1},
   |  |  |                g.switch_{k+1}, ..., g.switch_{g.switch.size()−1})
8  |  |  end
9  |  end
10 end
```

The final modification to the waveform is done by ELECTRICAL_MASKING, shown in Algorithm 16. The electrical properties of a gate remove short glitches, i.e., changes of the value that last for a short time only. A common abbreviation for this time is half the delay of gate. To remain conservative, we use half the minimal delay and set the variable t for the threshold accordingly in line 1. In the entwined loops from line 2 – 10, we check if two equal variables have a distance of t or less between them. If so, the waveform and the according switch times between the two variables describe a glitch that is removed. So we adjust the waveform and switch times of g accordingly in lines 6 and 7.

The effects of these algorithms are shown in Figure 4.8 to handle the same `nand`-gate that was handled in examples 4 to 9 for our monolithic approach.

4.3.4 The Algorithm GENERALIZE

The algorithm GENERALIZE shown in Algorithm 17 gets the assignment of a spurious counterexample and a primary output o that is affected by the SET according to the SAT formula. We use a greedy approach and return a SAT

Algorithm 17: GENERALIZE

input : a gate o that is a primary output, an assignment $a_{sim} : (I \cup C.G) \to \{0, 1, X\}^*$ of inputs and gate signals to values, a partial assignment $a_{SAT}(I \cup C.G) \to \{0, 1, X\}^*$, a circuit C and an SET s

output : a SAT formula f, when f is true, an eventual fault in o cannot propagate to the outputs of C

1 $FI :=$ const-fanin(g, a_{SAT})
2 sortByDistance(FI, o)
3 $a_{gen} := a_{gen} : FI \to \{0, 1, X\}$ with $a_{gen}(i) = a_{sim}(i)$
4 **foreach** $i \in \{FI.\mathit{size}(), \ldots, 1\}$ **do**
5 $\quad a_{gen}(FI_i) := X$
6 $\quad (a_{sim}, \mathit{real}) :=$ SIMULATE$(a_{gen}, C, s,$ fanin$(o))$
7 $\quad$ **if** *real* **then** $a_{gen}(FI_i) := a_{sim}(FI_i)$
8 **end**
9 **return** $\bigwedge_{\{g \in FI | a_{gen}(g) \neq X\}} g = a_{gen}(g)$

formula that describes a generalized assignment that suffices to prevent the SET from propagating towards o.

In line 1, we get the vector FI that contains the deepest constant signals within the fanin of o. We stop the search for the fanin at the first constant signal in the front partition according to a_{SAT} as these are equal in the assignment of the counterexample as well as the simulation because the different considerations of delays do not matter for constant signals. By this, we can further generalize the assignment. For example, in an `xor`-gate both inputs are relevant as a change of any input changes the output. However, we do not necessarily care about the exact inputs of the gate but only the output which can have different possible input assignments.

Afterwards, in line 2, we sort the vector FI by the distance of the gates to o. In this order, we can start to check gates that have a higher distance earlier and eventually set their assignment to X before checking closer gates that often have a higher impact on o. For example, an `or`-gate where one constant input is 1 only needs that 1 for its output to remain 1 and can set all variables that affect the other input to X.

In the loop from line 4 – 8, we try for each gate FI_i, in order from high to low distance to o, to set the assignment $a_{gen}(FI_i)$ to X in line 5 and simulate the modified assignment in line 6. To avoid unnecessary overhead, we only simulate the gates within the fanin of o. If the modified counterexample is real, i.e., o evaluates to X or the SET propagates to o, the value of FI_i is relevant for the SET not propagating towards o and we need to reset $a_{gen}(i)$ to its original value $a(i)$ in line 7.

The simulation is realized by an implementation similar to the algorithm CREATE_SAT. However, unlike CREATE_SAT, we compute the specific outputs under the given assignment. As only the output of o is relevant, we only simulate the fanout of o. An additional advantage of SIMULATE compared to CREATE_SAT is that we can easily consider the individual delays based on input values, as these values are known. Using this information in the SAT formula would complicate the SAT formula significantly.

Finally, in line 9 we return a formula that blocks the generalized counterexample.

4.3.5 Discussion

The hybrid approach provides a good performance due to two main reasons. On the one hand the generated SAT formula for the front partition is very simple and quickly solved and on the other hand our generalization allows us to block a high number of counterexamples after a single solver call.

When generating the SAT formula for the front partition, we can exploit the absence of a reconverging SET. Thus, we can describe the output of each gate with at most three variables as explained in Section 4.3.3. Additionally, no further variables are needed for the variable delays or electrical masking because it suffices to check for equal variables instead of equal values. The resulting SAT formula can usually solved within seconds or less and we can easily use the solver multiple times within a short time.

If we would block each spurious counterexample individually, the runtime would not be feasible for most circuits as there is usually a very high number of counterexamples. For this reason, we generalize counterexamples as shown in Section 4.3.4. By generalizing detected counterexamples, we can block multiple similar counterexamples with one SAT solver call. The degree of generalization depends on the circuit but usually provides a significant speed up.

4.4 Experiments

For our experiments, we use the ISCAS-85 benchmarks. To apply our algorithms to robust circuits, all circuits of the benchmarks have been modified into two robust versions. One version uses TMR to handle SETs. The original circuit is triplicated and a voter decides which output value is returned by using the value of the majority. The other version uses *Timed TMR* (TTMR) similar to [42]. The outputs of the original circuit are delayed by buffers. A voter decides similarly to TMR by using the current values, the value delayed by δ buffers and the value delayed by 2δ buffers. This method requires less overhead than TMR but still provides robustness against SETs as long as their duration is short enough.

We run the experiments on a Dual-Core AMD Opteron Processor 2222 SE with 3 GHz and 64 GB main memory. The transistor-level simulation is done with the tool Spectre from Cadence using a commercial 65nm technology.

The experiments are separated into three parts. In the first part, we compare our monolithic approach against Spectre to show that the models of our algorithms are correct and they conservatively check robustness. We also test how the conservatism affects the gained results.

In the second part, we will run our algorithms on the ISCAS-85 benchmarks to show their performance. Beside our two algorithms, we will also run experiments with a monolithic approach that uses the second model for signals, i.e., the model with switch times, to show that this model does speed up the algorithm.

Finally, we will analyze in high detail how the length of the SET and the grade of variability affects the runtime of our hybrid approach.

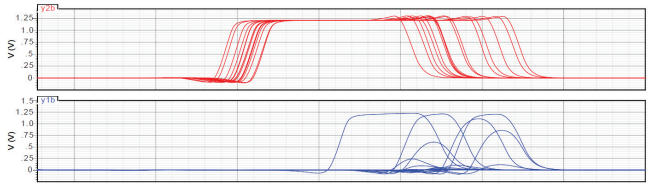


Figure 4.9: Output of c17 under a glitch in primary input G3

Assignments	Spectre	Our algorithm
G1 = 1, G3 = 1	82ps	30ps
G1 = 1, G3 = 0	111ps	60ps
G1 = 0, G3 = 1	107ps	60ps
G1 = 0, G3 = 0	76ps	60ps

Table 4.1: Minimal SET causing an error per input and method

4.4.1 Validation

To validate the accuracy and functionality of the monolithic approach we test it against Spectre. We use c17 as a test case and perform a detailed Monte-Carlo simulation including on-chip variability for each transistor. The effect of the alpha particles is modeled as a double exponential current pulse with a parameterizable energy, as done in [39]. When all the inputs are set to 1 and an SET is induced into input G3 the effect is clearly visible in one output as seen in the top graph in Figure 4.9. However, the effects on the other output vary depending on the variability of the gates as seen on the lower graph. Unlike previous symbolic tools that do not consider variability, our algorithms can discover the possible error on the second output.

Furthermore, we validate that the results of the monolithic approach and Spectre are consistent for any input combination. Because of the long simulation time required by Spectre, we disabled the variability analysis. In this second experiment we used c17 with TTMR, the SET is induced into G10, a `nand`-gate directly behind the inputs. Different strengths of particle strikes are simulated with Spectre for all possible input valuations. Due to the physical behavior the minimal strength of the particle strike and therefore the length of the SET that leads to an error differs depending on the inputs G1 and G3 of G10. The results of our algorithm depend on the expected output value of G10. Since there are two possible output values, we consider two different cases while Spectre considers four different cases, one for each possible input of G10. We adjusted our algorithm to return all counterexamples instead of one to check which assignments of variables are counterexamples for a given SET. We used timesteps of 5ps for our algorithm. The results of this experiment are shown in Table 4.1. Since the Spectre simulations did not consider variability in this experiment, we set $d_{min} = d_{max}$ for all gates. The result of our algorithm is usually off by 50ps due to abstractions from the transistor level and conservative analysis of electrical masking. Besides the different minimal lengths, our algorithm returned exactly all counterexamples that were confirmed by Spectre to lead to an error.

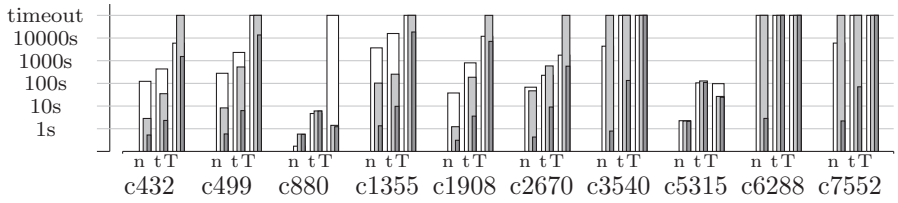


Figure 4.10: Runtime experiments comparing the monolithic approach (white), the monolithic approach with the switch times model (light gray), and our hybrid approach on ISCAS-85 circuits, using the normal (n), TMR (t), and TTMR (T) version

circuit	[58]	SAT-based	hybrid	solver calls
c432	123.1s	2.8s	0.5s	2
c432-TMR	432.5s	35.1s	2.3s	5
c432-TTMR	5943.4s	timeout	1534.1s	1342
c2670	67.8s	47s	0.4s	2
c2670-TMR	227.0s	588.9s	9.1s	3
c2670-TTMR	1758.2s	timeout	573.2s	1025
c7552	5982.3s	timeout	2.2s	1
c7552-TMR	timeout	timeout	70.7s	11
c7552-TTMR	timeout	timeout	timeout	2227

Table 4.2: Runtimes of all approaches and number of solver calls for hybrid approach on some circuits

The transistor-level simulation for all possible inputs and particle strikes on c17 took a few hours while our algorithm took less than a minute, which is a significant speedup in comparison to Spectre. As the model of the hybrid is equivalent to the model of the monolithic approach, the gained results can also be applied to the hybrid algorithm.

4.4.2 Runtime

In these experiments, we compare the runtimes our hybrid approach against our monolithic approach. In addition, we also consider a modified monolithic version that uses the second model for signals, i.e., the model with switch times. We use the ISCAS-85 benchmarks. Each circuit is analyzed in its regular version and in two fault tolerant variations. The first variation uses TMR and the second uses TTMR.

For each circuit, we choose a random gate near the inputs as location for the SET. An SET close to the inputs usually affects more gates and leads to a larger number of gates where the SET overlaps to sufficiently compare the differences between the algorithms.

The resulting runtimes are shown in Figure 4.10. In addition, a table with the runtimes on a number of chosen circuits that presents all approaches as well as the number of solver calls done by our hybrid approach is shown for some circuits in Table 4.2.

We can see that our hybrid approach is usually faster than the monolithic approach unless both time out after six hours. The modified monolithic approach often has a runtime between the two other approaches, showing that our model also decreases the runtime.

Even on the bigger circuits the hybrid algorithm takes less than 2.5 seconds to find a counterexample that disproves robustness for the normal non-robust versions. The hybrid algorithm generates a simple SAT formula for the front and finds counterexamples quickly. Since the circuits are not robust by themselves, only few counterexamples need to be simulated until a real counterexample is found. In comparison, the monolithic approach creates a complex SAT formula for each circuit which takes more time to solve.

For the robust circuits, the hybrid approach needs to generalize the detected counterexamples until robustness is proven. The generalization for the TMR circuits is quickly done as for each primary output the outputs of the two copies that are unaffected by the SET suffice to guarantee a correct value in the primary output. In both TMR and TTMR, we exploit that the fault correction is applied to each primary output individually and only need to analyze the relevant fanin.

In all TTMR circuits, the modified monolithic approach times out. Due to the higher degree of detail for delays, the different switch times overlap and new variables need to be introduced to describe the value in between. With increasing depth of the circuit, this effect leads to an exponential growth of variables for each gate which especially affects the TTMR circuits. This effect only occurs when the SET reconverges in the front partition and therefore does not affect our hybrid approach.

Only the circuits c880 and c5315 in the normal and TMR version are decided faster by [58]. In these cases, the location of the SET leads to a very small back partition. Thus, the resulting SAT formula for the front is only slightly easier to solve than the one generated by [58]. As our implementation may need multiple counterexamples even in non-robust circuits, our runtime is slightly higher in these specific cases. However, over all experiments the hybrid approach provides an average speedup of 748 compared to the monolithic approach.

Further experiments were done to check the significance of our optimizations. In our experiments with the monolithic approach, the runtime decreased significantly with each optimization, i.e., reusing variables, considering constant signals, and exploiting the form of the SET. Especially the optimization to consider constant signals provided an average speedup of 40.

We ran the experiments on c2670 with our hybrid approach again without using generalization. In the non-robust circuit, the number of detected spurious counterexamples only increased slightly as a real counterexample can easily be found in a small number of tries. In the robust cases, we generated 86 times more counterexamples and 340 times the previous runtime on average. These numbers show that the generalization is relevant for the runtime as discussed in Section 4.3.5.

As an alternative way to partition the circuit, we tried to put all gates within the back partition that require more than k variables for different k from 10 until all gates were in the front part. This alternate partitioning did not provide any significant speed up and took more time in most cases.

4.4.3 Effects of Variability and SET Duration

In these experiments, we will show the effects of different SET durations and variablility on the hybrid approach. As the monolithic approach is not that precise when describing delays, we do not consider it here.

We focus on the circuit c432 from the ISCAS'85 benchmarks. The original circuit c432 has 36 primary inputs, 7 primary outputs, and 160 gates. We consider the modified TTMR-version of c432 where every primary output is changed to handle short SETs. During our experiments with different parameters, we measure the runtime and the number of generated counterexamples for each run.

As location for the SET, we chose the gate U162 near the inputs. This choice lengthens the path of the SET to the primary inputs and increases the rate of reconvergence, increasing the effect of variability on the primary outputs. During our experiments, we use different numbers n of buffers to delay the output values, different grades of variability, and different durations of the SET. We run our experiments with $n = 1, 2, 3$ and a variability between 0ns and 0.1ns. The grade of variability describes the difference between minimum and maximum delay for each gate and each possible change of inputs. The duration of the SET is chosen depending on n, such that we show experiments near the critical duration where the circuit changes between being robust and not robust. Outside of the shown intervals, the runtime and number of counterexamples do not change any more, except for extremely short SETs.

The results of our experiments are shown in Tables 4.3, 4.4, and 4.5. Each table shows the experiments for a different number n of delaying buffers. Tables 4.3, 4.4, and 4.5 present the results for $n = 1$, $n = 2$, and $n = 3$, respectively. For each experiment, we show if the algorithm decides whether the circuit is robust or not, shown by the color. Red means that the circuit is not robust against the given SET under the considered variability and green means that it is robust. The first number shows the runtime of the experiment and the second one describes the number of spurious counterexamples that were generated during the run of the algorithm.

Since the initial SAT formula for the front area is almost identical for all experiments, the number of spurious counterexamples is approximately proportional to the runtime.

The critical duration of an SET, i.e., the shortest duration that can cause an error in the circuit, depends on the variability. Higher variability will decrease the critical duration as an error is more likely under high variation.

When increasing the duration of the SET above the highest or below the lowest shown value, the number of counterexamples and the duration remain the same. When the SET is long enough, it can propagate towards all primary outputs it can reach and is not prevented by TTMR or electrical masking. Choosing an even longer duration will only change the duration of an erroneous output value but not the existence of it. Therefore, a real counterexample can be found within a short runtime that does not depend upon the length of the SET or the grade of variability as long as the SET is long enough.

On the other hand, when the SET is sufficiently short, it will be blocked by TTMR at the same locations and the runtime will not change as long as the SET is not extremely short. In that case, it will be blocked by electrical masking alone and the runtime will decrease again.

SET \ Variability	0.1ns	0.05ns	0.025ns	0.01ns	0ns
3.3ns	1.46s 8cex	1.46s 8cex	1.48s 8cex	1.5s 8cex	1.52s 8cex
3.2ns	1.45s 8cex	1.51s 8cex	1.43s 8cex	6.69s 50cex	6.67s 50cex
3.1ns	1.45s 8cex	1.51s 8cex	1.99s 15cex	7.67s 52cex	7.61s 52cex
3ns	1.6s 8cex	2.11s 15cex	8.44s 15cex	8.36s 52cex	9.31s 55cex
2.9ns	1.48s 8cex	8.53s 62cex	7.86s 52cex	5.79s 54cex	5.75s 54cex
2.8ns	1.95s 15cex	7.08s 49cex	5.82s 56cex	6.08s 53cex	6.06s 53cex
2.7ns	2.01s 15cex	9.63s 62cex	6.07s 53cex	31.7s 150cex	31.69s 150cex
2.6ns	7.16s 49cex	6.89s 52cex	6.38s 55cex	137.87s 317cex	137.11s 317cex
2.5ns	9.34s 55cex	9.49s 76cex	382.89s 573cex	2.94s 21cex	156.27s 389cex
2.4ns	5.28s 46cex	23.05s 127cex	79.42s 244cex	87.28s 249cex	264.49s 512cex
2.3ns	10.04s 85cex	20.12s 117cex	2609.3s 1696cex	1637.73s 1368cex	1643.78s 1368cex
2.2ns	5.68s 46cex	7.22s 56cex	1571.19s 1409cex	1494.78s 1324cex	1488.91s 1324cex
2.1ns	5.54s 48cex	2291.56s 1628cex	1437.01s 1306cex	1464.18s 1325cex	1469.44s 1325cex
2ns	43.64s 196cex	1749.6s 1395cex	1443.78s 1265cex	1442.54s 1225cex	1444.62s 1225cex
1.9ns	24.99s 152cex	1427.54s 1296cex	1424.67s 1225cex	1432.54s 1225cex	1425.56s 1225cex
1.8ns	25.48s 152cex	1423.82s 1225cex	1429.58s 1225cex	1430.65s 1225cex	1432.03s 1225cex
1.7ns	2154.52s 1511cex	1435.44s 1225cex	1429.88s 1225cex	1431.15s 1225cex	1432.75s 1225cex
1.6ns	1526.2s 1335cex	1435.44s 1225cex	1429.88s 1225cex	1431.15s 1225cex	1432.75s 1225cex
1.5ns	1431.61s 1265cex	1435.44s 1225cex	1429.88s 1225cex	1431.15s 1225cex	1432.75s 1225cex
1.4ns	1436.94s 1225cex	1435.44s 1225cex	1429.88s 1225cex	1431.15s 1225cex	1432.75s 1225cex

Table 4.3: Experiments with c432-TTMR1 showing runtime, number of spurious counterexamples and whether the circuit is robust (green) or not robust (red)

SET \ Variability	0.1ns	0.05ns	0.025ns	0.01ns	0ns
6.3ns	1.59s 8cex	1.56s 8cex	1.6s 8cex	1.92s 8cex	1.56s 8cex
6.2ns	1.58s 8cex	1.58s 8cex	1.57s 8cex	7.01s 50cex	6.91s 50cex
6.1ns	1.62s 8cex	1.6s 8cex	2.1s 15cex	7.3s 49cex	9.38s 65cex
6ns	1.87s 8cex	2.53s 15cex	8.6s 51cex	10.49s 63cex	10.45s 63cex
5.9ns	1.62s 8cex	7.42s 51cex	9.38s 63cex	5.49s 51cex	5.55s 51cex
5.8ns	2.06s 15cex	9.05s 63cex	5.46s 51cex	6.28s 53cex	6.45s 53cex
5.7ns	2.04s 15cex	2.78s 25cex	6.37s 53cex	32.07s 150cex	32.63s 150cex
5.6ns	9.16s 63cex	3.35s 30cex	5.63s 46cex	140.11s 317cex	139.98s 317cex
5.5ns	4.92s 31cex	5.6s 46cex	77.29s 223cex	3.13s 21cex	3.03s 21cex
5.4ns	6.09s 42cex	12.29s 66cex	21.87s 113cex	159.96s 389cex	695.82s 856cex
5.3ns	2.67s 19cex	22.1s 113cex	3.11s 21cex	1605.97s 1376cex	1603.08s 1376cex
5.2ns	8.48s 51cex	3.11s 21cex	1639.44s 1358cex	1528.97s 1341cex	1534.24s 1341cex
5.1ns	16.6s 113cex	3.14s 21cex	1492.15s 1269cex	1142.26s 1306cex	1446.41s 1306cex
5ns	11.9s 71cex	1667.54s 1358cex	1472.64s 1306cex	1467.99s 1306cex	1471.21s 1306cex
4.9ns	3.04s 21cex	1533.06s 1341cex	1442.38s 1306cex	1443.93s 1306cex	1442.71s 1306cex
4.8ns	3.07s 21cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.7ns	3.19s 21cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.6ns	1444.49s 1306cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.5ns	1464.23s 1317cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.4ns	1449.39s 1306cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.3ns	1449.39s 1306cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex
4.2ns	1449.39s 1306cex	1445.28s 1306cex	1448.15s 1306cex	1447.78s 1306cex	1449.84s 1306cex

Table 4.4: Experiments with c432-TTMR2

SET \ Variability	0.1ns	0.05ns	0.025ns	0.01ns	0ns
	1.76s 8cex	1.77s 8cex	1.79s 8cex	1.77s 8cex	
15.2ns	1.78s 8cex	1.76s 8cex	1.77s 8cex	1.79s 8cex	1.77s 8cex
15.1ns	1.79s 8cex	1.78s 8cex	1.76s 8cex	8.3s 51cex	8.27s 51cex
15ns	1.81s 8cex	1.81s 8cex	8.2s 51cex	10.37s 63cex	10.22s 63cex
14.9ns	1.77s 8cex	2.28s 15cex	9.95s 63cex	5.81s 51cex	5.71s 51cex
14.8ns	1.79s 8cex	10.36s 63cex	5.88s 51cex	6.97s 53cex	7s 53cex
14.7ns	2.27s 15cex	12.39s 78cex	6.12s 43cex	56.71s 185cex	56.06s 185cex
14.6ns	2.26s 15cex	7.87s 54cex	5.85s 42cex	45.04s 158cex	48.63s 171cex
14.5ns	5.55s 31cex	3.52s 21cex	279.7s 473cex	145.03s 317cex	144.17s 317cex
14.4ns	6.24s 41cex	11.22s 55cex	110.06s 262cex	165.24s 389cex	167.05s 389cex
14.3ns	2.85s 19cex	29.19s 124cex	3.53s 21cex	1636.99s 1376cex	1615.81s 1376cex
14.2ns	8.28s 45cex	11.14s 71cex	2030.54s 1543cex	1578.31s 1357cex	1578.8s 1357cex
14.1ns	18.06s 104cex	3.55s 21cex	1618.85s 1376cex	1487.12s 1342cex	1490.97s 1342cex
14ns	15.93s 101cex	1649.25s 1358cex	1558.4s 1357cex	1494.72s 1342cex	1494.28s 1342cex
13.9ns	10.82s 71cex	1505.97s 1270cex	1497.17s 1342cex	1491.71s 1342cex	1500.2s 1342cex
13.8ns	3.51s 21cex	1439.45s 1305cex	1494.52s 1342cex	1492.26s 1342cex	1502.66s 1342cex
13.7ns	3.49s 21cex	1495.61s 1342cex	1494.98s 1342cex	1498s 1342cex	1501.91s 1342cex
13.6ns	1942.93s 1473cex	1495.61s 1342cex	1494.98s 1342cex	1498s 1342cex	1501.91s 1342cex
13.5ns	1470.27s 1332cex	1495.61s 1342cex	1494.98s 1342cex	1498s 1342cex	1501.91s 1342cex
13.4ns	1499.71s 1342cex	1495.61s 1342cex	1494.98s 1342cex	1498s 1342cex	1501.91s 1342cex

Table 4.5: Experiments with c432-TTMR3

The runtime usually increases the closer the duration of the SET is to the critical duration and timing becomes very important. In the non-robust case, the number of real counterexamples usually decreases. Thus, we detect more spurious ones until a real counterexample is detected. Since the SAT formula is more conservative than the simulation in respect to the timing, additional spurious counterexamples exist. Detecting these spurious counterexamples will increase the runtime.

However, in some experiments, the runtime decreases close to the critical duration. It is possible that a SAT solver detects a real counterexample earlier even though more spurious counterexamples exist due to its used heuristics. Another possibility can be signals that overlap in this specific setup and allow more generalization or decrease the number of spurious counterexamples. So, while the runtime usually increases close to the critical duration, some circumstances can also decrease it.

Compared to simulative approaches, the described algorithm can handle different grades of variability without a decrease in runtime by using three-valued logic. On the other hand, simulative approaches need to do multiple simulations to consider different variations in the delays. When the grade of variation is changed for a simulative approach, the number of required simulation runs increases exponentially to the grade of variation. As mentioned before, analyzing the circuit c17 of ISCAS-85 with 5 gates under a small grade of variability took hours while our algorithms could detect all real counterexamples to robustness within seconds.

We tried to run the equivalent experiments with the modified monolithic approach. When we consider the experiment on c432-TTMR3 with the duration 14.2ns of the SET and a grade 0.025ns of variability, the experiment took 2030s with the hybrid approach, generating 1543 spurious counterexamples. The initial SAT formula consisted of 6962 variables and grew to 904751 variables when we blocked all spurious counterexamples. On the other hand, the SAT-only approach was not even able to generate the complete SAT formula within six hours.

4.5 Conclusion

We presented two verification approaches considering SETs under logic, timing, and electrical masking, including variation, and considering all possible input assignments. Validation against transistor-level simulations shows the conservativeness. Runtimes cannot directly be compared with previous techniques as the approach has unique characteristics. However, the approach is significantly faster compared to simulating individual input assignments.

We presented a monolithic and a hybrid algorithm to decide if a circuit is robust against a given SET. The monolithic approach describes the whole circuit including the SET as a SAT formula and runs a SAT solver to prove robustness or return a counterexample that disproves robustness. The hybrid algorithm partitions the circuit into a front and a back partition, uses SAT solving on the front partition and analyzes detected counterexamples by simulation to refine a SAT formula until robustness can be decided. The experiments showed that dividing the problem this way can lead to a significant speed up.

Our approaches can be exploited for characterizing a design under variation effects and can be complemented by expensive Spice simulations.

Chapter 5

Outlook

While both contributed approaches are finished and can be used to solve the corresponding problems, expansions to increase the performance or widen the field of application are still possible.

A comparison between neighboring abstraction levels could provide further possibilities of application. An equivalence check between ESL and a HDL would allow further use of the equivalence checker as it could support the development of hardware systems further along the development cycle. However, both levels are quite different. While an execution on ESL coalues. Thus, the correspondence mapping that defines which initial states and which methods correspond to each other needs to be expanded. While defining corresponding initial states should be straightforward, as we just need to define thnsists of the execution of multiple methods, a system described in a HDL changes its state and outputs during each clock cycle according to its current state and input ve initial assignment of the registers in HDL, the function mapping needs to consider a number of methods on ESL and specific input values over a certain number of clock cycles for the HDL.

Enabling the equivalence check between ESL and HDL would also allow the use of the equivalence checker together with the robustness checker. In this scenario, a hardened iteration of the system at HDL can be checked for correctness by comparing it to the golden model at ESL. In a next step, tools can deduce the logical circuit from the HDL level and our robustness checker can check that circuit for robustness. If the equivalence check and the robustness check are successful, we have shown that the system at HDL is both correct and robust.

Bibliography

[1] Intel rechisels the tablet on moore's law. http://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/. Accessed: 2016-07-22.

[2] Systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94, Dec 2011.

[3] Bijan Alizadeh and Masahiro Fujita. Automatic merge-point detection for sequential equivalence checking of system-level and rtl descriptions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 129–144, 2007.

[4] S. Antunes. *Surviving Orbit the DIY Way: Testing the Limits Your Satellite Can and Must Match*. DIY satellites. O'Reilly Media, 2012.

[5] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[6] Soumyadip Bandyopadhyay, Dipankar Sarkar, Kunal Banerjee, and CA Mandal. A path-based equivalence checking method for petri net based models of programs. In *International Conference on Software Engineering and Applications*, pages 319–329, 2015.

[7] Aaron R. Bradley. SAT-based model checking without unrolling. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–87, 2011.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[9] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference*, pages 368–371, 2003.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[11] Mehdi Dehbashi and Görschwin Fey. SAT-based speedpath debugging using waveforms. In *IEEE European Test Symposium*, pages 1–6, 2014.

[12] R. Drechsler, M. Soeken, and R. Wille. Formal specification level: Towards verification-driven design based on natural language processing. In *Forum on specification and Design Languages*, pages 53–58, Sept 2012.

[13] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 125–134, 2011.

[14] Piet Engelke, Ilia Polian, Juergen Schloeffel, and Bernd Becker. Resistive bridging fault simulation of industrial circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 628–633, New York, NY, USA, 2008. ACM.

[15] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, 2003.

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, pages 35–45, 2007.

[17] Xiushan Feng and Alan J. Hu. Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification. In *Design Automation Conference*, pages 1063–1068, 2006.

[18] Alexander Finder, Jan-Philipp Witte, and Görschwin Fey. Debugging HDL designs based on functional equivalences with high-level specifications. In *International Symposium on Automated Technology for Verification and Analysis*, pages 60–65, 2013.

[19] Stefan Frehse, Görschwin Fey, Eli Arbel, Karen Yorav, and Rolf Drechsler. Complete and effective robustness checking by means of interpolation. In *Formal Methods in Computer-Aided Design*, pages 82–90, 2012.

[20] M. J. Gadlage, R. D. Schrimpf, J. M. Benedetto, P. H. Eaton, D. G. Mavis, M. Sibley, K. Avery, and T. L. Turflinger. Single event transient pulse widths in digital microcircuits. *IEEE Transactions on Nuclear Science*, 51(6):3285–3290, Dec 2004.

[21] Shanghua Gao, Takeshi Matsumoto, Hiroaki Yoshida, and Masahiro Fujita. Equivalence checking of loops before and after pipelining by applying symbolic simulation and induction. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 380–385, 2009.

[22] J.O. Grady. *System Validation and Verification*. Systems Engineering. Taylor & Francis, 1997.

[23] Ghaith Bany Hamad, Ghaith Khazma, Otmane Ait Mohamed, and Yvon Savaria. Comprehensive non-functional analysis of combinational circuits vulnerability of single event transients. In *Forum on specification and Design Languages*, pages 50–56, 2016.

[24] Jie Han, Hao Chen, Jinghang Liang, Peican Zhu, Zhixi Yang, and Fabrizio Lombardi. A stochastic computational approach for accurate and efficient reliability evaluation. *Computers, IEEE Transactions on*, pages 1336–1350, 2014.

[25] A. Holmes-Siedle and L. Adams. *Handbook of radiation effects*. Oxford science publications. Oxford University Press, 1993.

[26] D. Jurafsky, J.H. Martin, P. Norvig, and S. Russell. *Speech and Language Processing*. Pearson Education, 2014.

[27] J. Justesen and T. Høholdt. *A Course in Error-correcting Codes*. EMS textbooks in mathematics. European Mathematical Society, 2004.

[28] Kuk-Hwan Kim, Siddharth Gaba, Dana Wheeler, Jose M. Cruz-Albrecht, Tahir Hussain, Narayan Srinivasa, and Wei Lu. A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications. *Nano Letters*, 12(1):389–395, 2012. PMID: 22141918.

[29] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Design, Automation and Test in Europe*, pages 196–201, 2009.

[30] S. Krishnaswamy, I.L. Markov, and J.P. Hayes. *Design, Analysis and Test of Logic Circuits Under Uncertainty*. Lecture Notes in Electrical Engineering. Springer Netherlands, 2012.

[31] P. Lee. *Introduction to Place and Route Design in VLSIs*. Lulu.com, 2007.

[32] A. Leung, D. Bounov, and S. Lerner. C-to-verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 42–47, Sept 2015.

[33] Regis Leveugle. A new approach for early dependability evaluation based on formal property checking and controlled mutations. In *On-Line Testing Symposium*, pages 260–265, July 2005.

[34] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[35] W. Maly. Realistic fault modeling for vlsi testing. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pages 173–180, New York, NY, USA, 1987. ACM.

[36] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *Proceedings of the International Symposium on Quality Electronic Design*, pages 370–375, 2006.

[37] Natasa Miskov-Zivanov and Diana Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 1614–1627, 2010.

[38] Sparsh Mittal. A survey of architectural techniques for managing process variation. *ACM Comput. Surv.*, 48(4):54:1–54:29, February 2016.

[39] Kartik Mohanram. Simulation of transients caused by single-event upsets in combinational logic. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 9 pp.–981, 2005.

[40] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[41] Rajdeep Mukherjee, Daniel Kroening, Tom Melham, and Mandayam K.Srivas. Equivalence checking using trace partitioning. In *IEEE Computer Society Annual Symposium on VLSI*, pages 13–18, 2015.

[42] Michael Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *IEEE VLSI Test Symposium*, pages 86–94, 1999.

[43] Eugene Normand. Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[44] E. Notenboom. *Testing Embedded Software*. A Pearson education book. Addison-Wesley, 2003.

[45] Martin Omaña, Giacinto Papasso, Daniele Rossi, and Cecilia Metra. A model for transient fault propagation in combinatorial logic. In *IEEE International On-Line Testing Symposium*, pages 111–115, 2003.

[46] D. Perry and H. Foster. *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill electronic engineering series. McGraw-Hill Education, 2005.

[47] E. Petersen. *Single Event Effects in Aerospace*. Wiley, 2011.

[48] L.L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House computing library. Artech House, 2001.

[49] S. Rigo, R. Azevedo, and L. Santos. *Electronic System Level Design: An Open-Source Approach*. Springer Netherlands, 2011.

[50] M. O. Saglamdemir, G. Dundar, and A. Sen. An analog behavioral equivalence checking methodology for simulink models and circuit level designs. In *Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2015 International Conference on*, pages 1–4, Sept 2015.

[51] Matthias Sauer, Alexander Czutro, Ilia Polian, and Bernd Becker. Small-delay-fault ATPG with waveform accuracy. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–36, 2012.

[52] Ashwin Seshia, Wenchao Li, and S Mitra. Verification-guided soft error resilience. In *Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007.

[53] Kodamballi C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe*, pages 1310–1315, 2005.

[54] S.Z. Shazli and M.B. Tahoori. Using boolean satisfiability for computing soft error rates in early design stages. *Microelectronics Reliability*, 50(1):149–159, 2010.

[55] A. Sivadasan, F. Cacho, S. A. Benhassain, V. Huard, and L. Anghel. Study of workload impact on bti hci induced aging of digital circuits. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1020–1021, March 2016.

[56] A. Taber and E. Normand. Single event upset in avionics. *IEEE Transactions on Nuclear Science*, 40(2):120–126, Apr 1993.

[57] Niels Thole, Lorena Anghel, and Görschwin Fey. A hybrid algorithm to conservatively check the robustness of circuits. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2016.

[58] Niels Thole, Görschwin Fey, and Alberto Garcia-Ortiz. Conservatively analyzing transient faults. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2015.

[59] Niels Thole, Heinz Riener, and Görschwin Fey. Equivalence checking on system level using a priori knowledge. In *Proceedings of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pages 177–182, April 2015.

[60] Niels Thole, Heinz Riener, and Görschwin Fey. Equivalence checking on esl utilizing a priori knowledge. In *Forum on specification and Design Languages*, 2016.

[61] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL*. John Wiley & Sons, 2010.

[62] Grace Wu, Yi-Tin Sun, and Jie-Hong R. Jiang. Design partitioning for large-scale equivalence checking and functional correction. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 23:1–23:6, New York, NY, USA, 2016. ACM.

[63] Sheng Yu. *Handbook of Formal Languages: Volumne 1. Word, Languages, Grammar*, chapter Regular Languages, pages 41–110. Springer Science & Business Media, 1997.